عنوان: **EF Code First #1**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۱۴ ۱۰:۰۷:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Entity framework

در ادامه بحث ASP.NET MVC می‌شود به ابزاری به نام MVC Scaffolding اشاره کرد. کار این ابزار که توسط یکی از اعضای تیم ASP.NET MVC به نام استیو اندرسون تهیه شده، تولید کدهای اولیه یک برنامه کامل ASP.NET MVC از روی مدل‌های شما می‌باشد. حجم بالایی از کدهای تکراری آغازین برنامه را می‌شود توسط این ابزار تولید و بعد سفارشی کرد. MVC Scaffolding حتی قابلیت تولید کد بر اساس الگوی Repository و یا نوشتن Unit tests مرتبط را نیز دارد. بدیهی است این ابزار جای یک برنامه نویس را نمی‌تواند پر کند اما کدهای آغازین یک سری کارهای متداول و تکراری را به خوبی می‌تواند پیاده سازی و ارائه دهد. زیر ساخت این ابزار، علاوه بر ASP.NET MVC، آشنایی با Entity framework code first است.
در طی سری ASP.NET MVC که در این سایت تا به اینجا مطالعه کردید من به شدت سعی کردم از ابزارگرایی پرهیز کنم. چون شخصی که نمی‌داند مسیریابی چیست، اطلاعات چگونه به یک کنترلر منتقل یا به یک View ارسال می‌شوند، قراردادهای پیش فرض فریم ورک چیست یا زیر ساخت امنیتی یا فیلترهای ASP.NET MVC کدامند، چطور می‌تواند از ابزار پیشرفته Code generator ایی استفاده کند، یا حتی در ادامه کدهای تولیدی آن‌را سفارشی سازی کند؟ بنابراین برای استفاده از این ابزار و درک کدهای تولیدی آن، نیاز به یک پیشنیاز دیگر هم وجود دارد: «Entity framework code first»
امسال دو کتاب خوب در این زمینه منتشر شده‌اند به نام‌های:
Programming Entity Framework: DbContext , ISBN: 978-1-449-31296-1
Programming Entity Framework: Code First , ISBN: 978-1-449-31294-7
که هر دو به صورت اختصاصی به مقوله EF Code first پرداخته‌اند.

در طی روزهای بعدی EF Code first را با هم مرور خواهیم کرد و البته این مرور مستقل است از نوع فناوری میزبان آن؛ می‌خواهد یک برنامه کنسول باشد یا WPF یا یک سرویس ویندوز NT و یا ... یک برنامه وب.
البته از دیدگاه مایکروسافت، M در MVC به معنای EF Code first است. به همین جهت MVC3 به صورت پیش فرض ارجاعی را به اسمبلی‌های آن دارد و یا حتی به روز رسانی که برای آن ارائه داده نیز در جهت تکمیل همین بحث است.

مروری سریع بر تاریخچه Entity framework code first

ویژوال استودیو 2010 و دات نت 4، به همراه EF 4.0 ارائه شدند. با این نگارش امکان استفاده از حالت‌های طراحی database first و model first مهیا است. پس از آن، به روز رسانی‌های EF خارج از نوبت و به صورت منظم، هر از چندگاهی ارائه می‌شوند و در زمان نگارش این مطلب، آخرین نگارش پایدار در دسترس آن 4.3.1 می‌باشد. از زمان EF 4.1 به بعد، نوع جدیدی از مدل سازی به نام Code first به این فریم ورک اضافه شد و در نگارش‌های بعدی آن، مباحث DB migration جهت ساده سازی تطابق اطلاعات مدل‌ها با بانک اطلاعاتی، اضافه گردیدند. در روش Code first، کار با طراحی کلاس‌ها که در اینجا مدل داده‌ها نامیده می‌شوند، شروع گردیده و سپس بر اساس این اطلاعات، تولید یک بانک اطلاعاتی جدید و یا استفاده از نمونه‌ای موجود میسر می‌گردد.
پیشتر در روش database first ابتدا یک بانک اطلاعاتی موجود، مهندسی معکوس می‌شد و از روی آن فایل XML ایی با پسوند EDMX تولید می‌گشت. سپس به کمک entity data model designer ویژوال استودیو، این فایل نمایش داده شده و یا امکان اعمال تغییرات بر روی آن میسر می‌شد. همچنین در روش دیگری به نام model first نیز کار از entity data model designer جهت طراحی موجودیت‌ها آغاز می‌گشت.
اما با روش Code first دیگر در ابتدای امر مدل فیزیکی و یک بانک اطلاعاتی وجود خارجی ندارد. در اینجا EF تعاریف کلاس‌های شما را بررسی کرده و بر اساس آن، اطلاعات نگاشت‌های خواص کلاس‌ها به جداول و فیلدهای بانک اطلاعاتی را تشکیل می‌دهد. البته عموما تعاریف ساده کلاس‌ها بر این منظور کافی نیستند. به همین جهت از یک سری متادیتا به نام ویژگی‌ها یا اصطلاحا data annotations مهیا در فضای نام System.ComponentModel.DataAnnotations برای افزودن اطلاعات لازم مانند نام فیلدها، جداول و یا تعاریف روابط ویژه نیز استفاده می‌گردد. به علاوه در روش Code first یک API جدید به نام Fluent API نیز جهت تعاریف این

ویژگی‌ها و روابط، با کدنویسی مستقیم نیز درنظر گرفته شده است. نهایتا از این اطلاعات جهت نگاشت کلاس‌ها به بانک اطلاعاتی و یا برای تولید ساختار یک بانک اطلاعاتی خالی جدید نیز می‌توان کمک گرفت.

مزایای EF Code first

- مطلوب برنامه نویس‌ها! : برنامه نویس‌هایی که مدتی تجربه کار با ابزارهای طراح را داشته باشند به خوبی می‌دانند این نوع ابزارها عموما demo-ware هستند. چندجا کلیک می‌کنید، دوبار Next، سه بار OK و ... به نظر می‌رسد کار تمام شده. اما واقعیت این است که عمری را باید صرف نگهداری و یا پیاده سازی جزئیاتی کرد که انجام آن‌ها با کدنویسی مستقیم بسیار سریعتر، ساده‌تر و با کنترل بیشتری قابل انجام است.
- سرعت: برای کار با EF Code first نیازی نیست در ابتدای کار بانک اطلاعاتی خاصی وجود داشته باشد. کلاس‌های خود را طراحی و شروع به کدنویسی کنید.
- سادگی: در اینجا دیگر از فایل‌های EDMX خبری نیست و نیازی نیست مرتبا آن‌ها را به روز کرده یا نگهداری کرد. تمام کارها را با کدنویسی و کنترل بیشتری می‌توان انجام داد. به علاوه کنترل کاملی بر روی کد نهایی تهیه شده نیز وجود دارد و توسط ابزارهای تولید کد، ایجاد نمی‌شوند.
- طراحی بهتر بانک اطلاعاتی نهایی: اگر طرح دقیقی از مدل‌های برنامه داشته باشیم، می‌توان آن‌ها را به المان‌های کوچک و مشخصی، تقسیم و refactor کرد. همین مساله در نهایت مباحث database normalization را به نحوی مطلوب و با سرعت بیشتری میسر می‌کند.
- امکان استفاده مجدد از طراحی کلاس‌های انجام شده در سایر ORMهای دیگر. چون طراحی مدل‌های برنامه به بانک اطلاعاتی خاصی گره نمی‌خورند و همچنین الزاما هم قرار نیست جزئیات کاری EF در آن‌ها لحاظ شود، این کلاس‌ها در صورت نیاز در سایر پروژه‌ها نیز به سادگی قابل استفاده هستند.
- ردیابی ساده‌تر تغییرات: روش اصولی کار با پروژه‌های نرم افزاری همواره شامل استفاده از یک ابزار سورس کنترل مانند SVN، Git، مرکوریال و امثال آن است. به این ترتیب ردیابی تغییرات انجام شده به سادگی توسط این ابزارها میسر می‌شوند.
- ساده‌تر شدن طراحی‌های پیچیده‌تر: برای مثال پیاده سازی ارث بری، ایجاد کلاس‌های خود ارجاع دهنده و امثال آن با کدنویسی ساده‌تر است.

دریافت آخرین نگارش EF

برای دریافت و نصب آخرین نگارش EF نیاز است از NuGet استفاده شود و این مزایا را به همراه دارد:
به کمک NuGet امکان با خبر شدن از به روز رسانی جدید صورت گرفته به صورت خودکار درنظر گرفته شده است و همچنین کار دریافت بسته‌های مرتبط و به روز رسانی ارجاعات نیز در این حالت خودکار است. به علاوه توسط NuGet امکان دسترسی به کتابخانه‌هایی که مثلا در گوگل‌کد قرار دارند و به صورت معمول امکان دریافت آن‌ها برای ما میسر نیست، نیز بدون مشکل فراهم است (برای نمونه ELMAH، که اصل آن از گوگل‌کد قابل دریافت است؛ اما بسته نیوگت آن نیز در دسترس می‌باشد).
پس از نصب NuGet، تنها کافی است بر روی گره References در Solution explorer ویژوال استودیو، کلیک راست کرده و به کمک NuGet آخرین نگارش EF را نصب کرد. در گالری آنلاین آن، عموما EF اولین گزینه است (به علت تعداد بالای دریافت آن).
حین استفاده از NuGet جهت نصب Ef، ابتدا ارجاعاتی به اسمبلی‌های زیر به برنامه اضافه خواهند شد:

System.ComponentModel.DataAnnotations.dll

System.Data.Entity.dll

EntityFramework.dll

بدیهی است بدون استفاده از NuGet، تمام این کارها را باید دستی انجام داد.
سپس در پوشه‌ای به نام packages، فایل‌های مرتبط با EF قرار خواهند گرفت که شامل اسمبلی آن به همراه ابزارهای DB Migration است. همچنین فایل packages.config که شامل تعاریف اسمبلی‌های نصب شده است به پروژه اضافه می‌شود. NuGet به کمک این فایل و شماره نگارش درج شده در آن، شما را از به روز رسانی‌های بعدی مطلع خواهد ساخت:

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
  <package id="EntityFramework" version="4.3.1" />
</packages>
```

همچنین اگر به فایل app.config یا web.config برنامه نیز مراجعه کنید، یک سری تنظیمات ابتدایی اتصال به بانک اطلاعاتی در آن ذکر شده است:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <!-- For more information on Entity Framework configuration, visit
http://go.microsoft.com/fwlink/?LinkID=237468 -->
    <section name="entityFramework"
type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework, Version=4.3.1.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089" />
  </configSections>
  <entityFramework>
    <defaultConnectionFactory type="System.Data.Entity.Infrastructure.SqlConnectionFactory,
EntityFramework">
      <parameters>
        <parameter value="Data Source=(localdb)\v11.0; Integrated Security=True;
MultipleActiveResultSets=True" />
      </parameters>
    </defaultConnectionFactory>
  </entityFramework>
</configuration>
```

همانطور که ملاحظه می‌کنید بانک اطلاعاتی پیش فرضی که در اینجا ذکر شده است، LocalDB می‌باشد. این بانک اطلاعاتی را از این آدرس نیز می‌توانید دریافت کنید.

البته ضرورتی هم به استفاده از آن نیست و از سایر نگارش‌های SQL Server نیز می‌توان استفاده کرد ولی خوب ... مزیت استفاده از آن برای کاربر نهایی این است که «نیازی به یک مهندس برای نصب، راه اندازی و نگهداری ندارد». تنها مشکل آن این است که از ویندوز XP پشتیبانی نمی‌کند. البته SQL Server CE 4.0 این محدودیت را ندارد.
ضمن اینکه باید درنظر داشت EF به فناوری میزبان خاصی گره نخورده است و مثال‌هایی که در اینجا بررسی می‌شوند صرفا تعدادی برنامه کنسول معمولی هستند و نکات عنوان شده در آن‌ها در تمام فناوری‌های میزبان موجود به یک نحو کاربرد دارند.

قراردادهای پیش فرض EF Code first

عنوان شد که اطلاعات کلاس‌های ساده تشکیل دهنده مدل‌های برنامه، برای تعریف جداول و فیلدهای یک بانک اطلاعات و همچنین مشخص سازی روابط بین آن‌ها کافی نیستند و مرسوم است برای پر کردن این خلاء از یک سری متادیتا و یا Fluent API مهیا نیز استفاده گردد. اما در EF Code first یک سری قرار داد توکار نیز وجود دارند که مطلع بودن از آن‌ها سبب خواهد شد تا حجم کدنویسی و تنظیمات جانبی این فریم ورک به حداقل برسند. برای نمونه مدل‌های معروف بلاگ و مطالب آن‌را درنظر بگیرید:

```
namespace EF_Sample01.Models
{
    public class Post
    {
        public int Id { set; get; }
        public string Title { set; get; }
        public string Content { set; get; }
        public virtual Blog Blog { set; get; }
    }
}
```

```
using System.Collections.Generic;

namespace EF_Sample01.Models
{
```

```
    public class Blog
    {
        public int Id { set; get; }
        public string Title { set; get; }
        public string AuthorName { set; get; }
        public IList<Post> Posts { set; get; }
    }
}
```

یکی از قراردادهای EF Code first این است که کلاس‌های مدل شما را جهت یافتن خاصیتی به نام Id یا ClassId مانند BlogId، جستجو می‌کند و از آن به عنوان primary key و فیلد identity جدول بانک اطلاعاتی استفاده خواهد کرد.
همچنین در کلاس Blog، خاصیت لیستی از Posts و در کلاس Post خاصیت virtual ایی به نام Blog وجود دارند. به این ترتیب روابط بین دو کلاس و ایجاد کلید خارجی متناظر با آن‌را به صورت خودکار انجام خواهد داد.
نهایتا از این اطلاعات جهت تشکیل database schema یا ساختار بانک اطلاعاتی استفاده می‌گردد.
اگر به فضاهای نام دو کلاس فوق دقت کرده باشید، به کلمه Models ختم شده‌اند. به این معنا که در پوشه‌ای به همین نام در پروژه جاری قرار دارند. یا مرسوم است کلاس‌های مدل را در یک پروژه class library مجزا به نام DomainClasses نیز قرار دهند. این پروژه نیازی به ارجاعات اسمبلی‌های EF ندارد و تنها به اسمبلی System.ComponentModel.DataAnnotations.dll نیاز خواهد داشت.

EF Code first چگونه کلاس‌های مورد نظر را انتخاب می‌کند؟

ممکن است ده‌ها و صدها کلاس در یک پروژه وجود داشته باشند. EF Code first چگونه از بین این کلاس‌ها تشخیص خواهد داد که باید از کدامیک استفاده کند؟ اینجا است که مفهوم جدیدی به نام DbContext معرفی شده است. برای تعریف آن یک کلاس دیگر را به پروژه برای مثال به نام Context اضافه کنید. همچنین مرسوم است که این کلاس را در پروژه class library دیگری به نام DataLayer اضافه می‌کنند. این پروژه نیاز به ارجاعی به اسمبلی‌های EF خواهد داشت. در ادامه کلاس جدید اضافه شده باید از کلاس DbContext مشتق شود:

```
using System.Data.Entity;
using EF_Sample01.Models;

namespace EF_Sample01
{
    public class Context : DbContext
    {
        public DbSet<Blog> Blogs { set; get; }
        public DbSet<Post> Posts { set; get; }
    }
}
```

سپس در اینجا به کمک نوع جنریکی به نام DbSet، کلاس‌های دومین برنامه را معرفی می‌کنیم. به این ترتیب، EF Code first ابتدا به دنبال کلاسی مشتق شده از DbContext خواهد گشت. پس از یافتن آن، خواصی از نوع DbSet را بررسی کرده و نوع‌های متناظر با آن‌را به عنوان کلاس‌های دومین درنظر می‌گیرد و از سایر کلاس‌های برنامه صرفنظر خواهد کرد. این کل کاری است که باید انجام شود.
اگر دقت کرده باشید، نام کلاس‌های موجودیت‌ها، مفرد هستند و نام خواص تعریف شده به کمک DbSet، جمع می‌باشند که نهایتا متناظر خواهند بود با نام جداول بانک اطلاعاتی تشکیل شده.

تشکیل خودکار بانک اطلاعاتی و افزودن اطلاعات به جداول

تا اینجا بدون تهیه یک بانک اطلاعاتی نیز می‌توان از کلاس Context تهیه شده استفاده کرد و کار کدنویسی را آغاز نمود. بدیهی

است جهت اجرای نهایی کدها، نیاز به یک بانک اطلاعاتی خواهد بود. اگر تنظیمات پیش فرض فایل کانفیگ برنامه را تغییر ندهیم، از همان defaultConnectionFactory یاده شده استفاده خواهد کرد. در این حالت نام بانک اطلاعاتی به صورت خودکار تنظیم شده و مساوی «EF_Sample01.Context» خواهد بود.
برای سفارشی سازی آن نیاز است فایل app.config یا web.config برنامه را اندکی ویرایش نمود:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
...
  </configSections>
  <connectionStrings>
    <clear/>
    <add name="Context"
         connectionString="Data Source=(local);Initial Catalog=testdb2012;Integrated Security = true"
          providerName="System.Data.SqlClient"
           />
  </connectionStrings>
...
</configuration>
```

در اینجا به بانک اطلاعاتی testdb2012 در وهله پیش فرض SQL Server نصب شده، اشاره شده است. فقط باید دقت داشت که تگ configSections باید در ابتدای فایل قرار گیرد و مابقی تنظیمات پس از آن.
یا اگر علاقمند باشید که از SQL Server CE استفاده کنید، تنظیمات رشته اتصالی را به نحو زیر مقدار دهی نمائید:

```
<connectionStrings>
              <add name="MyContextName"
                         connectionString="Data Source=|DataDirectory|\Store.sdf"
                         providerName="System.Data.SqlServerCe.4.0" />
</connectionStrings>
```

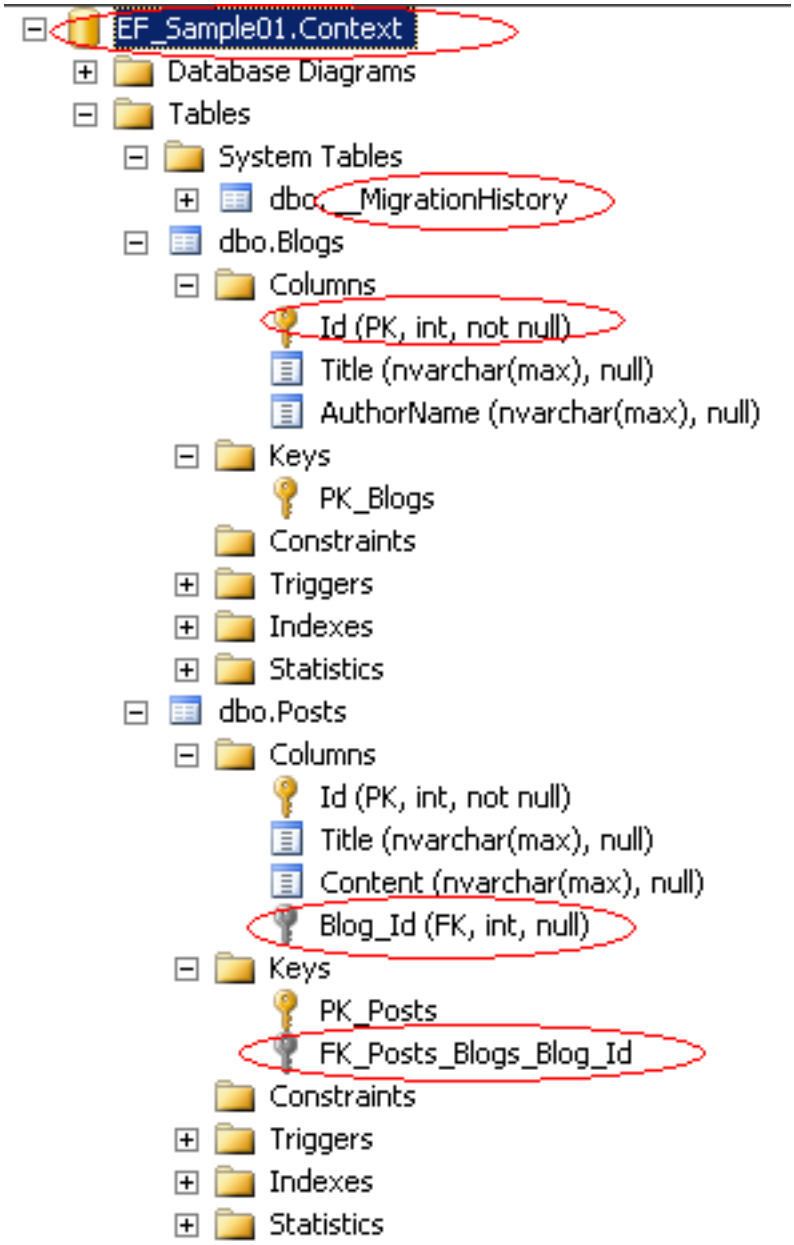
در هر دو حالت، name باید به نام کلاس مشتق شده از DbContext اشاره کند که در مثال جاری همان Context است.
یا اگر علاقمند بودید که این قرارداد توکار را تغییر داده و نام رشته اتصالی را با کدنویسی تعیین کنید، می‌توان به نحو زیر عمل کرد:

```
public class Context : DbContext
{
    public Context()
      : base("ConnectionStringName")
    {
    }
```

البته ضرورتی ندارد این بانک اطلاعاتی از پیش موجود باشد. در اولین بار اجرای کدهای زیر، به صورت خودکار بانک اطلاعاتی و جداول Blogs و Posts و روابط بین آن‌ها تشکیل می‌گردد:

```
using EF_Sample01.Models;

namespace EF_Sample01
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
            using (var db = new Context())
            {
                db.Blogs.Add(new Blog { AuthorName = "Vahid", Title = ".NET Tips" });
                db.SaveChanges();
            }
        }
    }
}
```



شکل ۱

در این تصویر چند نکته حائز اهمیت هستند:

الف) نام پیش فرض بانک اطلاعاتی که به آن اشاره شد (اگر تنظیمات رشته اتصالی قید نگردد).

ب) تشکیل خودکار primary key از روی خواصی به نام Id

ج) تشکیل خودکار روابط بین جداول و ایجاد کلید خارجی (به کمک خاصیت virtual تعریف شده)

د) تشکیل جدول سیستمی به نام dbo.__MigrationHistory که از آن برای نگهداری سابقه به روز رسانی‌های ساختار جداول کمک گرفته خواهد شد.

۵) نوع و طول فیلدهای متنی، nvarchar از نوع max است.

تمام این‌ها بر اساس پیش فرض‌ها و قراردادهای توکار EF Code first انجام شده است.

در کدهای تعریف شده نیز، ابتدا یک وهله از شیء Context ایجاد شده و سپس به کمک آن می‌توان به جدول Blogs اطلاعاتی را افزود و در آخر ذخیره نمود. استفاده از using هم دراینجا نباید فراموش شود، زیرا اگر استثنایی در این بین رخ دهد، کار پاکسازی منابع و بستن اتصال گشوده شده به بانک اطلاعاتی به صورت خودکار انجام خواهد شد.
در ادامه اگر بخواهیم مطلبی را به Blog ثبت شده اضافه کنیم، خواهیم داشت:

```
using EF_Sample01.Models;

namespace EF_Sample01
{
    class Program
    {
        static void Main(string[] args)
        {
            //addBlog();
            addPost();
        }

        private static void addPost()
        {
            using (var db = new Context())
            {
                var blog = db.Blogs.Find(1);
                db.Posts.Add(new Post
                {
                    Blog = blog,
                    Content = "data",
                    Title = "EF"
                });
                db.SaveChanges();
            }
        }

        private static void addBlog()
        {
            using (var db = new Context())
            {
                db.Blogs.Add(new Blog { AuthorName = "Vahid", Title = ".NET Tips" });
                db.SaveChanges();
            }
        }
    }
}
```

متد db.Blogs.Find، بر اساس primary key بلاگ ثبت شده، یک وهله از آن‌را یافته و سپس از آن جهت تشکیل شیء Post و افزودن آن به جدول Posts استفاده می‌شود. متد Find ابتدا Contxet جاری را جهت یافتن شیءایی با id مساوی یک جستجو می‌کند (اصطلاحا به آن first level cache هم گفته می‌شود). اگر موفق به یافتن آن شد، بدون صدور کوئری اضافه‌ای به بانک اطلاعاتی از اطلاعات همان شیء استفاده خواهد کرد. در غیراینصورت نیاز خواهد داشت تا ابتدا کوئری لازم را به بانک اطلاعاتی ارسال کرده و اطلاعات شیء Blog متناظر با id=1 را دریافت کند. همچنین اگر نیاز داشتیم تا تنها با سطح اول کش کار کنیم، در EF Code first می‌توان از خاصیتی به نام Local نیز استفاده کرد. برای مثال خاصیت db.Blogs.Local بیانگر اطلاعات موجود در سطح اول کش می‌باشد.
نهایتا کوئری Insert تولید شده توسط آن به شکل زیر خواهد بود (لاگ شده توسط برنامه SQL Server Profiler):

```
exec sp_executesql N'insert [dbo].[Posts]([Title], [Content], [Blog_Id])
values (@0, @1, @2)
select [Id]
from [dbo].[Posts]
where @@ROWCOUNT > 0 and [Id] = scope_identity()',
```

```
N'@0 nvarchar(max) ,@1 nvarchar(max) ,@2 int',
@0=N'EF',
@1=N'data',
@2=1
```

این نوع کوئرهای پارامتری چندین مزیت مهم را به همراه دارند:

الف) به صورت خودکار تشکیل می‌شوند. تمام کوئری‌های پشت صحنه EF پارامتری هستند و نیازی نیست مرتبا مزایای این امر را گوشزد کرد و باز هم عده‌ای با جمع زدن رشته‌ها نسبت به نوشتن کوئری‌های نا امن SQL اقدام کنند.

ب) کوئرهای پارامتری در مقابل حملات تزریق اس کیوال مقاوم هستند.

ج) SQL Server با کوئری‌های پارامتری همانند رویه‌های ذخیره شده رفتار می‌کند. یعنی query execution plan محاسبه شده آن‌ها را کش خواهد کرد. همین امر سبب بالا رفتن کارآیی برنامه در فراخوانی‌های بعدی می‌گردد. الگوی کلی مشخص است. فقط پارامترهای آن تغییر می‌کنند.

د) مصرف حافظه SQL Server کاهش می‌یابد. چون SQL Server مجبور نیست به ازای هر کوئری اصطلاحا Ad Hoc رسیده یکبار execution plan متفاوت آن‌ها را محاسبه و سپس کش کند. این مورد مشکل مهم تمام برنامه‌هایی است که از کوئری‌های پارامتری استفاده نمی‌کنند؛ تا حدی که گاهی تصور می‌کنند شاید SQL Server دچار نشتی حافظه شده، اما مشکل جای دیگری است.

مشکل! ساختار بانک اطلاعاتی تشکیل شده مطلوب کار ما نیست.

تا همینجا با حداقل کدنویسی و تنظیمات مرتبط با آن، پیشرفت خوبی داشته‌ایم؛ اما نتیجه حاصل آنچنان مطلوب نیست و نیاز به سفارشی سازی دارد. برای مثال طول فیلدها را نیاز داریم به مقدار دیگری تنظیم کنیم، تعدادی از فیلدها باید به صورت not null تعریف شوند یا نام پیش فرض بانک اطلاعاتی باید مشخص گردد و مواردی از این دست. با این موارد در قسمت‌های بعدی بیشتر آشنا خواهیم شد.

عنوان: EF Code First #2
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۱۵ ۰۹:۳۶:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Entity framework

در قسمت قبل با تنظیمات و قراردادهای ابتدایی EF Code first آشنا شدیم، هرچند این تنظیمات حجم کدنویسی ابتدایی راه اندازی سیستم را به شدت کاهش می‌دهند، اما کافی نیستند. در این قسمت نگاهی سطحی و مقدماتی خواهیم داشت بر امکانات مهیا جهت تنظیم ویژگی‌های مدل‌های برنامه در EF Code first.

تنظیمات EF Code first توسط اعمال متادیتای خواص

اغلب متادیتای مورد نیاز جهت اعمال تنظیمات EF Code first در اسمبلی System.ComponentModel.DataAnnotations.dll قرار دارند. بنابراین اگر مدل‌های خود را در اسمبلی و پروژه class library جداگانه‌ای تعریف و نگهداری می‌کنید (مثلا به نام DomainClasses)، نیاز است ابتدا ارجاعی را به این اسمبلی به پروژه جاری اضافه نمائیم. همچنین تعدادی دیگر از متادیتای قابل استفاده در خود اسمبلی EntityFramework.dll قرار دارند. بنابراین در صورت نیاز باید ارجاعی را به این اسمبلی نیز اضافه نمود.
همان مثال قبل را در اینجا ادامه می‌دهیم. دو کلاس Blog و Post در آن تعریف شده (به این نوع کلاس‌ها POCO – the Plain Old CLR Objects نیز گفته می‌شود)، به همراه کلاس Context که از کلاس DbContext مشتق شده است. ابتدا دیتابیس قبلی را دستی drop کنید. سپس در کلاس Blog، خاصیت public int Id را مثلا به public int MyTableKey تغییر دهید و پروژه را اجرا کنید. برنامه بلافاصله با خطای زیر متوقف می‌شود:

```
One or more validation errors were detected during model generation:
\tSystem.Data.Entity.Edm.EdmEntityType: : EntityType 'Blog' has no key defined.
```

زیرا EF Code first در این کلاس خاصیتی به نام Id یا BlogId را نیافته‌است و امکان تشکیل Primary key جدول را ندارد. برای رفع این مشکل تنها کافی است ویژگی Key را به این خاصیت اعمال کنیم:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace EF_Sample01.Models
{
    public class Blog
    {
        [Key]
        public int MyTableKey { set; get; }
```

همچنین تعدادی ویژگی دیگر مانند MaxLength و Required را نیز می‌توان بر روی خواص کلاس اعمال کرد:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace EF_Sample01.Models
{
    public class Blog
    {
        [Key]
        public int MyTableKey { set; get; }

        [MaxLength(100)]
```

```
        public string Title { set; get; }

        [Required]
        public string AuthorName { set; get; }

        public IList<Post> Posts { set; get; }
    }
}
```

این ویژگی‌ها دو مقصود مهم را برآورده می‌سازند:
الف) بر روی ساختار بانک اطلاعاتی تشکیل شده تاثیر دارند:

```
CREATE TABLE [dbo].[Blogs](
 [MyTableKey] [int] IDENTITY(1,1) NOT NULL,
 [Title] [nvarchar](100) NULL,
 [AuthorName] [nvarchar](max) NOT NULL,
 CONSTRAINT [PK_Blogs] PRIMARY KEY CLUSTERED
(
 [MyTableKey] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY]
```

همانطور که ملاحظه می‌کنید در اینجا طول فیلد Title به 100 تنظیم شده است و همچنین فیلد AuthorName اینبار NOT NULL است. به علاوه primary key نیز بر اساس ویژگی Key اعمالی تعیین شده است.
البته برای اجرای کدهای تغییر کرده مدل، فعلا بانک اطلاعاتی قبلی را دستی می‌توان حذف کرد تا بتوان به ساختار جدید رسید. در مورد جزئیات مبحث DB Migration در قسمت‌های بعدی مفصلا بحث خواهد شد.

ب) اعتبار سنجی اطلاعات پیش از ارسال کوئری به بانک اطلاعاتی
برای مثال اگر در حین تعریف وهله‌ای از کلاس Blog، خاصیت AuthorName مقدار دهی نگردد، پیش از اینکه رفت و برگشتی به بانک اطلاعاتی صورت گیرد، یک validation error را دریافت خواهیم کرد. یا برای مثال اگر طول اطلاعات خاصیت Title بیش از 100 حرف باشد نیز مجددا در حین ثبت اطلاعات، یک استثنای اعتبار سنجی را مشاهده خواهیم کرد. البته امکان تعریف پیغام‌های خطای سفارشی نیز وجود دارد. برای این حالت تنها کافی است پارامتر ErrorMessage این ویژگی‌ها را مقدار دهی کرد. برای مثال:

```
[Required(ErrorMessage = "لطفا نام نویسنده را مشخص نمائید")]
public string AuthorName { set; get; }
```

نکته‌ی مهمی که در اینجا وجود دارد، وجود یک اکوسیستم هماهنگ و سازگار است. این نوع اعتبار سنجی هم با EF Code first هماهنگ است و هم برای مثال در ASP.NET MVC به صورت خودکار جهت اعتبار سنجی سمت سرور و کلاینت یک مدل می‌تواند مورد استفاده قرار گیرد و مفاهیم و روش‌های مورد استفاده در آن نیز یکی است.

تنظیمات EF Code first به کمک Fluent API

اگر علاقمند به استفاده از متادیتا، جهت تعریف قیود و ویژگی‌های خواص کلاس‌های مدل خود نیستید، روش دیگری نیز در EF Code first به نام Fluent API تدارک دیده شده است. در اینجا امکان تعریف همان ویژگی‌ها توسط کدنویسی نیز وجود دارد، به علاوه اعمال قیود دیگری که توسط متادیتای مهیا قابل تعریف نیستند.
محل تعریف این قیود، کلاس Context که از کلاس DbContext مشتق شده است، می‌باشد و در اینجا، کار با تحریف متد OnModelCreating شروع می‌شود:

```
using System.Data.Entity;
using EF_Sample01.Models;

namespace EF_Sample01
{
    public class Context : DbContext
    {
        public DbSet<Blog> Blogs { set; get; }
        public DbSet<Post> Posts { set; get; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Blog>().HasKey(x => x.MyTableKey);
            modelBuilder.Entity<Blog>().Property(x => x.Title).HasMaxLength(100);
            modelBuilder.Entity<Blog>().Property(x => x.AuthorName).IsRequired();

            base.OnModelCreating(modelBuilder);
        }
    }
}
```

به کمک پارامتر modelBuilder، امکان دسترسی به متدهای تنظیم کننده ویژگی‌های خواص یک مدل یا موجودیت وجود دارد. در اینجا چون می‌توان متدها را به صورت یک زنجیره به هم متصل کرد و همچنین حاصل نهایی شبیه به جمله بندی انگلیسی است، به آن Fluent API یا API روان نیز گفته می‌شود.

البته در این حالت امکان تعریف ErrorMessage وجود ندارد و برای این منظور باید از همان data annotations استفاده کرد.

نحوه مدیریت صحیح تعاریف نگاشت‌ها به کمک Fluent API

OnModelCreating محل مناسبی جهت تعریف حجم انبوهی از تنظیمات کلاس‌های مختلف مدل‌های برنامه نیست. در حد سه چهار سطر مشکلی ندارد اما اگر بیشتر شد بهتر است از روش زیر استفاده شود:

```
using System.Data.Entity;
using EF_Sample01.Models;
using System.Data.Entity.ModelConfiguration;

namespace EF_Sample01
{
    public class BlogConfig : EntityTypeConfiguration<Blog>
    {
        public BlogConfig()
        {
            this.Property(x => x.Id).HasColumnName("MyTableKey");
            this.Property(x => x.RowVersion).HasColumnType("Timestamp");
        }
    }
```

با ارث بری از کلاس EntityTypeConfiguration، می‌توان به ازای هر کلاس مدل، تنظیمات را جداگانه انجام داد. به این ترتیب اصل SRP یا Single responsibility principle نقض نخواهد شد. سپس برای استفاده از این کلاس‌های Config تک مسئولیتی به نحو زیر می‌توان اقدام کرد:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
     modelBuilder.Configurations.Add(new BlogConfig());
```

نحوه تنظیمات ابتدایی نگاشت کلاس‌ها به بانک اطلاعاتی در EF Code first

الزامی ندارد که EF Code first حتما با یک بانک اطلاعاتی از نو تهیه شده بر اساس پیش فرض‌های آن کار کند. در اینجا می‌توان از بانک‌های اطلاعاتی موجود نیز استفاده کرد. اما در این حالت نیاز خواهد بود تا مثلا نام جدولی خاص با کلاسی مفروض در برنامه، یا نام فیلدی خاص که مطابق استانداردهای نامگذاری خواص در سی شارپ تعریف نشده، با خاصیتی در یک کلاس تطابق داده شوند. برای مثال اینبار تعاریف کلاس Blog را به نحو زیر تغییر دهید:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace EF_Sample01.Models
{
    [Table("tblBlogs")]
    public class Blog
    {
        [Column("MyTableKey")]
        public int Id { set; get; }

        [MaxLength(100)]
        public string Title { set; get; }

        [Required(ErrorMessage = "لطفا نام نویسنده را مشخص نمائید")]
        public string AuthorName { set; get; }

        public IList<Post> Posts { set; get; }

        [Timestamp]
        public byte[] RowVersion { set; get; }
    }
}
```

در اینجا فرض بر این است که نام جدول متناظر با کلاس Blog در بانک اطلاعاتی مثلا tblBlogs است و نام خاصیت Id در بانک اطلاعاتی مساوی فیلدی است به نام MyTableKey. چون نام خاصیت را مجددا به Id تغییر داده‌ایم، دیگر ضرورتی به ذکر ویژگی Key وجود نداشته است. برای تعریف این دو از ویژگی‌های Table و Column جهت سفارشی سازی نام‌های خواص و کلاس استفاده شده است.

یا اگر در کلاس خود خاصیتی محاسبه شده بر اساس سایر خواص، تعریف شده است و قصد نداریم آن‌را به فیلدی در بانک اطلاعاتی نگاشت کنیم، می‌توان از ویژگی NotMapped برای مزین سازی و تعریف آن کمک گرفت.

به علاوه اگر از نام پیش فرض کلید خارجی تشکیل شده خرسند نیستید می‌توان به کمک ویژگی ForeignKey، نسبت به تعریف مقداری جدید مطابق تعاریف یک بانک اطلاعاتی موجود، اقدام کرد.

همچنین خاصیت دیگری به نام RowVersion در اینجا اضافه شده که با ویژگی TimeStamp مزین گردیده است. از این خاصیت ویژه برای بررسی مسایل همزمانی ثبت اطلاعات در EF استفاده می‌شود. به علاوه بانک اطلاعاتی می‌تواند به صورت خودکار آن‌را در حین ثبت مقدار دهی کند.

تمام این تغییرات را به کمک Fluent API نیز می‌توان انجام داد:

```
modelBuilder.Entity<Blog>().ToTable("tblBlogs");
modelBuilder.Entity<Blog>().Property(x => x.Id).HasColumnName("MyTableKey");
modelBuilder.Entity<Blog>().Property(x => x.RowVersion).HasColumnType("Timestamp");
```

تبدیل پروژه‌های قدیمی EF به کلاس‌های EF Code first به صورت خودکار

روش متداول کار با EF از روز اول آن، مهندسی معکوس خودکار اطلاعات یک بانک اطلاعاتی و تبدیل آن به یک فایل EDMX بوده است. هنوز هم می‌توان از این روش در اینجا نیز بهره جست. برای مثال اگر قصد دارید یک پروژه قدیمی را تبدیل به نمونه جدید

Code first کنید، یا یک بانک اطلاعاتی موجود را مهندسی معکوس کنید، بر روی پروژه در Solution explorer کلیک راست کرده و گزینه Add|New Item را انتخاب کنید. سپس از صفحه ظاهر شده، ADO.NET Entity data model را انتخاب کرده و در ادامه گزینه «Generate from database» را انتخاب کنید. این روال مرسوم کار با EF Database first است.
پس از اتمام کار به entity data model designer مراجعه کرده و بر روی صفحه کلیک راست نمائید. از منوی ظاهر شده گزینه «Add code generation item» را انتخاب کنید. سپس در صفحه باز شده از لیست قالب‌های موجود، گزینه «ADO.NET DbContext Generator» را انتخاب نمائید. این گزینه به صورت خودکار اطلاعات فایل EDMX قدیمی یا موجود شما را تبدیل به کلاس‌های مدل Code first معادل به همراه کلاس DbContext معرف آن‌ها خواهد کرد.

روش دیگری نیز برای انجام اینکار وجود دارد. نیاز است افزونه‌ی به نام Entity Framework Power Tools را دریافت کنید. پس از نصب، از منوی Entity Framework آن گزینه‌ی «Reverse Engineer Code First» را انتخاب نمائید. در اینجا می‌توان مشخصات اتصال به بانک اطلاعاتی را تعریف و سپس نسبت به تولید خودکار کدهای مدل‌ها و DbContext مرتبط اقدام کرد.

استراتژی‌های مقدماتی تشکیل بانک اطلاعاتی در EF Code first

اگر مثال این سری را دنبال کرده باشید، مشاهده کرده‌اید که با اولین بار اجرای برنامه، یک بانک اطلاعاتی پیش فرض نیز تولید خواهد شد. یا اگر تعاریف ویژگی‌های یک فیلد را تغییر دادیم، نیاز است تا بانک اطلاعاتی را دستی drop کرده و اجازه دهیم تا بانک اطلاعاتی جدیدی بر اساس تعاریف جدید مدل‌ها تشکیل شود که ... هیچکدام از این‌ها بهینه نیستند.
در اینجا دو استراتژی مقدماتی را در حین آغاز یک برنامه می‌توان تعریف کرد:

```
System.Data.Entity.Database.SetInitializer(new DropCreateDatabaseIfModelChanges<Context>());
// or
System.Data.Entity.Database.SetInitializer(new DropCreateDatabaseAlways<Context>());
```

می‌توان بانک اطلاعاتی را در صورت تغییر اطلاعات یک مدل به صورت خودکار drop کرده و نسبت به ایجاد نمونه‌ای جدید اقدام کرد (DropCreateDatabaseIfModelChanges)؛ یا در حین آزمایش برنامه همیشه (DropCreateDatabaseAlways) با شروع برنامه، ابتدا باید بانک اطلاعاتی drop شده و سپس نمونه جدیدی تولید گردد.
محل فراخوانی این دستور هم باید در نقطه آغازین برنامه، پیش از وهله سازی اولین DbContext باشد. مثلا در برنامه‌های وب در متد Application_Start فایل global.asax.cs یا در برنامه‌های WPF در متد سازنده کلاس App می‌توان بانک اطلاعاتی را آغاز نمود.
البته الزامی به استفاده از کلاس‌های DropCreateDatabaseIfModelChanges یا DropCreateDatabaseAlways وجود ندارد. می‌توان با پیاده سازی اینترفیس IDatabaseInitializer از نوع کلاس Context تعریف شده در برنامه، همان عملیات را شبیه سازی کرد یا سفارشی نمود:

```
public class MyInitializer : IDatabaseInitializer<Context>
{
        public void InitializeDatabase(Context context)
        {
            if (context.Database.Exists() ||
                context.Database.CompatibleWithModel(throwIfNoMetadata: false))
                context.Database.Delete();

            context.Database.Create();
        }
}
```

سپس برای استفاده از این کلاس در ابتدای برنامه، خواهیم داشت:

```
System.Data.Entity.Database.SetInitializer(new MyInitializer());
```

نکته:
اگر از یک بانک اطلاعاتی موجود استفاده می‌کنید (محیط کاری) و نیازی به پیش فرض‌های EF Code first ندارید و همچنین این بانک اطلاعاتی نیز نباید drop شود یا تغییر کند، می‌توانید تمام این پیش فرض‌ها را با دستور زیر غیرفعال کنید:

```
Database.SetInitializer<Context>(null);
```

بدیهی است این دستور نیز باید پیش از ایجاد اولین وهله از شیء DbContext فراخوانی شود.

همچنین باید درنظر داشت که در آخرین نگارش‌های پایدار EF Code first، این موارد بهبود یافته‌اند و مبحثی تحت عنوان DB Migration ایجاد شده است تا نیازی نباشد هربار بانک اطلاعاتی drop شود و تمام اطلاعات از دست برود. می‌توان صرفا تغییرات کلاس‌ها را به بانک اطلاعاتی اعمال کرد که به صورت جداگانه، در قسمتی مجزا بررسی خواهد شد. به این ترتیب دیگر نیازی به drop بانک اطلاعاتی نخواهد بود. به صورت پیش فرض در صورت از دست رفتن اطلاعات یک استثناء را سبب خواهد شد (که توسط برنامه نویس قابل تنظیم است) و در حالت خودکار یا دستی با تنظیمات ویژه قابل اعمال است.

تنظیم استراتژی‌های آغاز بانک اطلاعاتی در فایل کانفیگ برنامه

الزامی ندارد که حتما متد Database.SetInitializer را دستی فراخوانی کنیم. با اندکی تنظیم فایل‌های app.config و یا web.config نیز می‌توان نوع استراتژی مورد استفاده را تعیین کرد:

```
<appSettings>
 <add key="DatabaseInitializerForType MyNamespace.MyDbContextClass, MyAssembly"
     value="MyNamespace.MyInitializerClass, MyAssembly"  />
</appSettings>

<appSettings>
 <add key="DatabaseInitializerForType MyNamespace.MyDbContextClass, MyAssembly"
     value="Disabled" />
</appSettings>
```

یکی از دو حالت فوق باید در قسمت appSettings فایل کانفیگ برنامه تنظیم شود. حالت دوم برای غیرفعال کردن پروسه آغاز بانک اطلاعاتی و اعمال تغییرات به آن، بکار می‌رود.
برای نمونه در مثال جاری، جهت استفاده از کلاس MyInitializer فوق، می‌توان از تنظیم زیر نیز استفاده کرد:

```
<appSettings>
    <add key="DatabaseInitializerForType EF_Sample01.Context, EF_Sample01"
       value="EF_Sample01.MyInitializer, EF_Sample01" />
  </appSettings>
```

اجرای کدهای ویژه در حین تشکیل یک بانک اطلاعاتی جدید

امکان سفارشی سازی این آغاز کننده‌های پیش فرض نیز وجود دارد. برای مثال:

```
public class MyCustomInitializer : DropCreateDatabaseIfModelChanges<Context>
{
    protected override void Seed(Context context)
    {
        context.Blogs.Add(new Blog { AuthorName = "Vahid", Title = ".NET Tips" });
        context.Database.ExecuteSqlCommand("CREATE INDEX IX_title ON tblBlogs (title)");
        base.Seed(context);
    }
}
```

در اینجا با ارث بری از کلاس DropCreateDatabaseIfModelChanges یک آغاز کننده سفارشی را تعریف کرده‌ایم. سپس با تحریف متد Seed آن می‌توان در حین آغاز یک بانک اطلاعاتی، تعدادی رکورد پیش فرض را به آن افزود. کار ذخیره سازی نهایی در متد base.Seed انجام می‌شود.

برای استفاده از آن اینبار در حین فراخوانی متد System.Data.Entity.Database.SetInitializer، از کلاس MyCustomInitializer استفاده خواهیم کرد.

و یا توسط متد context.Database.ExecuteSqlCommand می‌توان دستورات SQL را مستقیما در اینجا اجرا کرد. عموما دستوراتی در اینجا مدنظر هستند که توسط ORMها پشتیبانی نمی‌شوند. برای مثال تغییر collation یک ستون یا افزودن یک ایندکس و مواردی از این دست.

سطح دسترسی مورد نیاز جهت فراخوانی متد Database.SetInitializer

استفاده از متدهای آغاز کننده بانک اطلاعاتی نیاز به سطح دسترسی بر روی بانک اطلاعاتی master را در SQL Server دارند (زیرا با انجام کوئری بر روی این بانک اطلاعاتی مشخص می‌شود، آیا بانک اطلاعاتی مورد نظر پیشتر تعریف شده است یا خیر). البته این مورد حین کار با SQL Server CE شاید اهمیتی نداشته باشد. بنابراین اگر کاربری که با آن به بانک اطلاعاتی متصل می‌شویم سطح دسترسی پایینی دارد نیاز است Persist Security Info=True را به رشته اتصالی اضافه کرد. البته این مورد را پس از انجام تغییرات بر روی بانک اطلاعاتی جهت امنیت بیشتر حذف کنید (یا به عبارتی در محیط کاری Persist Security Info=False باید باشد).

```
Server=(local);Database=yourDatabase;User
ID=yourDBUser;Password=yourDBPassword;Trusted_Connection=False;Persist Security Info=True
```

تعیین Schema و کاربر فراخوان دستورات SQL

در EF Code first به صورت پیش فرض همه چیز بر مبنای کاربری با دسترسی مدیریتی یا dbo schema در اس کیوال سرور تنظیم شده است. اما اگر کاربر خاصی برای کار با دیتابیس تعریف گردد که در هاست‌های اشتراکی بسیار مرسوم است، دیگر از دسترسی مدیریتی dbo خبری نخواهد بود. اینبار نام جداول ما بجای dbo.tableName مثلا someUser.tableName می‌باشند و عدم دقت به این نکته، اجرای برنامه را غیرممکن می‌سازد.

برای تغییر و تعیین صریح کاربر متصل شده به بانک اطلاعاتی اگر از متادیتا استفاده می‌کنید، روش زیر باید بکارگرفته شود:

```
[Table("tblBlogs", Schema="someUser")]
public class Blog
```

و یا در حالت بکارگیری Fluent API به نحو زیر قابل تنظیم است:

```
modelBuilder.Entity<Blog>().ToTable("tblBlogs", schemaName:"someUser");
```

عنوان: EF Code First #3
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۱۶ ۱۷:۳۷:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Entity framework

بررسی تعاریف نگاشت‌ها به کمک متادیتا در EF Code first

در قسمت قبل مروری سطحی داشتیم بر امکانات مهیای جهت تعاریف نگاشت‌ها در EF Code first. در این قسمت، حالت استفاده از متادیتا یا همان data annotations را با جزئیات بیشتری بررسی خواهیم کرد.
برای این منظور پروژه کنسول جدیدی را آغاز نمائید. همچنین به کمک NuGet، ارجاعات لازم را به اسمبلی EF، اضافه کنید. در ادامه مدل‌های زیر را به پروژه اضافه نمائید؛ یک شخص که تعدادی پروژه منتسب می‌تواند داشته باشد:

```
using System;
using System.Collections.Generic;

namespace EF_Sample02.Models
{
    public class User
    {
        public int Id { set; get; }
        public DateTime AddDate { set; get; }
        public string Name { set; get; }
        public string LastName { set; get; }
        public string Email { set; get; }
        public string Description { set; get; }
        public byte[] Photo { set; get; }
        public IList<Project> Projects { set; get; }
    }
}
```

```
using System;

namespace EF_Sample02.Models
{
    public class Project
    {
        public int Id { set; get; }
        public DateTime AddDate { set; get; }
        public string Title { set; get; }
        public string Description { set; get; }
        public virtual User User { set; get; }
    }
}
```

به خاصیت public virtual User User در کلاس Project اصطلاحا Navigation property هم گفته می‌شود.
دو کلاس زیر را نیز جهت تعریف کلاس Context که بیانگر کلاس‌های شرکت کننده در تشکیل بانک اطلاعاتی هستند و همچنین کلاس آغاز کننده بانک اطلاعاتی سفارشی را به همراه تعدادی رکورد پیش فرض مشخص می‌کنند، به پروژه اضافه نمائید.

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using EF_Sample02.Models;

namespace EF_Sample02
{
    public class Sample2Context : DbContext
    {
```

```
        public DbSet<User> Users { set; get; }
        public DbSet<Project> Projects { set; get; }
    }

    public class Sample2DbInitializer : DropCreateDatabaseAlways<Sample2Context>
    {
        protected override void Seed(Sample2Context context)
        {
            context.Users.Add(new User
            {
                AddDate = DateTime.Now,
                Name = "Vahid",
                LastName = "N.",
                Email = "name@site.com",
                Description = "-",
                Projects = new List<Project>
                {
                    new Project
                    {
                        Title = "Project 1",
                        AddDate = DateTime.Now.AddDays(-10),
                        Description = "..."
                    }
                }
            });

            base.Seed(context);
        }
    }
}
```

به علاوه در فایل کانفیگ برنامه، تنظیمات رشته اتصالی را نیز اضافه نمائید:

```
<connectionStrings>
    <add
       name="Sample2Context"
       connectionString="Data Source=(local);Initial Catalog=testdb2012;Integrated Security = true"
       providerName="System.Data.SqlClient"
      />
  </connectionStrings>
```

همانطور که ملاحظه می‌کنید، در اینجا name به نام کلاس مشتق شده از DbContext اشاره می‌کند (یکی از قراردادهای توکار EF Code first است).

یک نکته:

مرسوم است کلاس‌های مدل را در یک class library جداگانه اضافه کنند به نام DomainClasses و کلاس‌های مرتبط با DbContext را در پروژه class library دیگری به نام DataLayer. هیچکدام از این پروژه‌ها نیازی به فایل کانفیگ و تنظیمات رشته اتصالی ندارند؛ زیرا اطلاعات لازم را از فایل کانفیگ پروژه اصلی که این دو پروژه class library را به خود الحاق کرده، دریافت می‌کنند. دو پروژه class library اضافه شده تنها باید ارجاعاتی را به اسمبلی‌های EF و data annotations داشته باشند.

در ادامه به کمک متد Database.SetInitializer که در قسمت دوم به بررسی آن پرداختیم و با استفاده از کلاس سفارشی Sample2DbInitializer فوق، نسبت به ایجاد یک بانک اطلاعاتی خالی تشکیل شده بر اساس تعاریف کلاس‌های دومین پروژه، اقدام خواهیم کرد:

```
using System;
using System.Data.Entity;

namespace EF_Sample02
{
    class Program
```

```
    {
        static void Main(string[] args)
        {
            Database.SetInitializer(new Sample2DbInitializer());
            using (var db = new Sample2Context())
            {
                var project1 = db.Projects.Find(1);
                Console.WriteLine(project1.Title);
            }
        }
    }
}
```

تا زمانیکه وهله‌ای از Sample2Context ساخته نشود و همچنین یک کوئری نیز به بانک اطلاعاتی ارسال نگردد، Sample2DbInitializer در عمل فراخوانی نخواهد شد.

ساختار بانک اطلاعاتی پیش فرض تشکیل شده نیز مطابق اسکریپت زیر است:

```
CREATE TABLE [dbo].[Users](
 [Id] [int] IDENTITY(1,1) NOT NULL,
 [AddDate] [datetime] NOT NULL,
 [Name] [nvarchar](max) NULL,
 [LastName] [nvarchar](max) NULL,
 [Email] [nvarchar](max) NULL,
 [Description] [nvarchar](max) NULL,
 [Photo] [varbinary](max) NULL,
 CONSTRAINT [PK_Users] PRIMARY KEY CLUSTERED
(
 [Id] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
 IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY]
```

```
CREATE TABLE [dbo].[Projects](
 [Id] [int] IDENTITY(1,1) NOT NULL,
 [AddDate] [datetime] NOT NULL,
 [Title] [nvarchar](max) NULL,
 [Description] [nvarchar](max) NULL,
 [User_Id] [int] NULL,
 CONSTRAINT [PK_Projects] PRIMARY KEY CLUSTERED
(
 [Id] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF,
IGNORE_DUP_KEY = OFF, ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY]

GO

ALTER TABLE [dbo].[Projects]  WITH CHECK ADD  CONSTRAINT [FK_Projects_Users_User_Id] FOREIGN
KEY([User_Id])
REFERENCES [dbo].[Users] ([Id])
GO

ALTER TABLE [dbo].[Projects] CHECK CONSTRAINT [FK_Projects_Users_User_Id]
GO
```

توضیحاتی در مورد ساختار فوق، جهت یادآوری مباحث دو قسمت قبل:

- خواصی با نام Id تبدیل به primary key و identity field شده‌اند.
- نام جداول، همان نام خواص تعریف شده در کلاس Context است.
- تمام رشته‌ها به nvarchar از نوع max نگاشت شده‌اند و null پذیر می‌باشند.
- خاصیت تصویر که با آرایه‌ای از بایت‌ها تعریف شده به varbinary از نوع max نگاشت شده است.
- بر اساس ارتباط بین کلاس‌ها فیلد User_Id در جدول Projects اضافه شده است که توسط قیدی به نام FK_Projects_Users_User_Id، جهت تعریف کلید خارجی عمل می‌کند. این نام گذاری پیش فرض هم بر اساس نام خواص در دو

کلاس انجام می‌شود.
- schema پیش فرض بکارگرفته شده، dbo است.
- null پذیری پیش فرض فیلدها بر اساس اصول زبان مورد استفاده تعیین شده است. برای مثال در سی شارپ، نوع int نال پذیر نیست یا نوع DateTime نیز به همین ترتیب یک value type است. بنابراین در اینجا این دو نوع به صورت not null تعریف شده‌اند (صرفنظر از اینکه در SQL Server هر دو نوع یاد شده، null پذیر هم می‌توانند باشند). بدیهی است امکان تعریف nullable types نیز وجود دارد.

مروری بر انواع متادیتای قابل استفاده در EF Code first

1) Key

همانطور که ملاحظه کردید اگر نام خاصیتی Id یا ClassName+Id باشد، به صورت خودکار به عنوان primary key جدول، مورد استفاده قرار خواهد گرفت. این یک قرارداد توکار است.
اگر یک چنین خاصیتی با نام‌های ذکر شده در کلاس وجود نداشته باشد، می‌توان با مزین سازی خاصیتی مفروض با ویژگی Key که در فضای نام System.ComponentModel.DataAnnotations قرار دارد، آن‌را به عنوان Primary key معرفی نمود. برای مثال:

```
public class Project
{
     [Key]
     public int ThisIsMyPrimaryKey { set; get; }
```

و ضمنا باید دقت داشت که حین کار با ORMs فرقی نمی‌کند EF باشد یا سایر فریم ورک‌های دیگر، داشتن یک key جهت عملکرد صحیح فریم ورک، ضروری است. بر اساس یک Key است که Entity معنا پیدا می‌کند.

2) Required

ویژگی Required که در فضای نام System.ComponentModel.DataAnnotations تعریف شده است، سبب خواهد شد یک خاصیت به صورت not null در بانک اطلاعاتی تعریف شود. همچنین در مباحث اعتبارسنجی برنامه، پیش از ارسال اطلاعات به سرور نیز نقش خواهد داشت. در صورت نال بودن خاصیتی که با ویژگی Required مزین شده است، یک استثنای اعتبارسنجی پیش از ذخیره سازی اطلاعات در بانک اطلاعاتی صادر می‌گردد. این ویژگی علاوه بر EF Code first در ASP.NET MVC نیز به نحو یکسانی تاثیرگذار است.

3) MaxLength و MinLength

این دو ویژگی نیز در فضای نام System.ComponentModel.DataAnnotations قرار دارند (اما در اسمبلی EntityFramework.dll تعریف شده‌اند و جزو اسمبلی پایه System.ComponentModel.DataAnnotations.dll نیستند). در ذیل نمونه‌ای از تعریف این‌ها را مشاهده می‌کنید. همچنین باید درنظر داشت که روش دیگر تعریف متادیتا، ترکیب آن‌ها در یک سطر نیز می‌باشد. یعنی الزامی ندارد در هر سطر یک متادیتا را تعریف کرد:

```
[MaxLength(50, ErrorMessage = "حداکثر 50 حرف"), MinLength(4, ErrorMessage = "حداقل 4 حرف")]
public string Title { set; get; }
```

ویژگی MaxLength بر روی طول فیلد تعریف شده در بانک اطلاعاتی تاثیر دارد. برای مثال در اینجا فیلد Title از نوع nvarchar با طول 30 تعریف خواهد شد.
ویژگی MinLength در بانک اطلاعاتی معنایی ندارد.
هر دوی این ویژگی‌ها در پروسه اعتبار سنجی اطلاعات مدل دریافتی تاثیر دارند. برای مثال در اینجا اگر طول عنوان کمتر از 4 حرف باشد، یک استثنای اعتبارسنجی صادر خواهد شد.

ویژگی دیگری نیز به نام StringLength وجود دارد که جهت تعیین حداکثر طول رشته‌ها به کار می‌رود. این ویژگی سازگاری بیشتر با ASP.NET MVC دارد از این جهت که Client side validation آن‌را نیز فعال می‌کند.

4) Table و Column

این دو ویژگی نیز در فضای نام System.ComponentModel.DataAnnotations قرار دارند، اما در اسمبلی EntityFramework.dll تعریف شده‌اند. بنابراین اگر تعاریف مدل‌های شما در پروژه Class library جداگانه‌ای قراردارند، نیاز خواهد بود تا ارجاعی را به اسمبلی EntityFramework.dll نیز داشته باشند.

اگر از نام پیش فرض جداول تشکیل شده خرسند نیستید، ویژگی Table را بر روی یک کلاس قرار داده و نام دیگری را تعریف کنید. همچنین اگر Schema کاربری رشته اتصالی به بانک اطلاعاتی شما dbo نیست، باید آن‌را در اینجا صریحا ذکر کنید تا کوئری‌های تشکیل شده به درستی بر روی بانک اطلاعاتی اجرا گردند:

```
[Table("tblProject", Schema="guest")]
public class Project
```

توسط ویژگی Column سه خاصیت یک فیلد بانک اطلاعاتی را می‌توان تعیین کرد:

```
[Column("DateStarted", Order = 4, TypeName = "date")]
public DateTime AddDate { set; get; }
```

به صورت پیش فرض، خاصیت فوق با همین نام AddDate در بانک اطلاعاتی ظاهر می‌گردد. اگر برای مثال قرار است از یک بانک اطلاعاتی قدیمی استفاده شود یا قرار نیست از شیوه نامگذاری خواص در سی شارپ در یک بانک اطلاعاتی پیروی شود، توسط ویژگی Column می‌توان این تعاریف را سفارشی نمود.

توسط پارامتر Order آن که از صفر شروع می‌شود، ترتیب قرارگیری فیلدها در حین تشکیل یک جدول مشخص می‌گردد.

اگر نیاز است نوع فیلد تشکیل شده را نیز سفارشی سازی نمائید، می‌توان از پارامتر TypeName استفاده کرد. برای مثال در اینجا علاقمندیم از نوع date مهیا در SQL Server 2008 استفاده کنیم و نه از نوع datetime پیش فرض آن.

نکته‌ای در مورد Order:

Order پیش فرض تمام خواصی که قرار است به بانک اطلاعاتی نگاشت شوند، به int.MaxValue تنظیم شده‌اند. به این معنا که تنظیم فوق با Order=4 سبب خواهد شد تا این فیلد، پیش از تمام فیلدهای دیگر قرار گیرد. بنابراین نیاز است Order اولین خاصیت تعریف شده را به صفر تنظیم نمود. (البته اگر واقعا نیاز به تنظیم دستی Order داشتید)

نکاتی در مورد تنظیمات ارث بری در حالت استفاده از متادیتا:

حداقل سه حالت ارث بری را در EF code first می‌توان تعریف و مدیریت کرد:

الف) Table per Hierarchy - TPH

حالت پیش فرض است. نیازی به هیچگونه تنظیمی ندارد. معنای آن این است که «لطفا تمام اطلاعات کلاس‌هایی را که از هم ارث بری کرده‌اند در یک جدول بانک اطلاعاتی قرار بده». فرض کنید یک کلاس پایه شخص را دارید که کلاس‌های بازیکن و مربی از آن ارث بری می‌کنند. زمانیکه کلاس پایه شخص توسط DbSet در کلاس مشتق شده از DbContext در معرض استفاده EF قرار می‌گیرد، بدون نیاز به هیچ تنظیمی، تمام این سه کلاس، تبدیل به یک جدول شخص در بانک اطلاعاتی خواهند شد. یعنی یک table به ازای سلسله مراتبی (Hierarchy) که تعریف شده.

ب) Table per Type - TPT

به این معنا است که به ازای هر نوع، باید یک جدول تشکیل شود. به عبارتی در مثال قبل، یک جدول برای شخص، یک جدول برای مربی و یک جدول برای بازیکن تشکیل خواهد شد. دو جدول مربی و بازیکن با یک کلید خارجی به جدول شخص مرتبط می‌شوند.

تنها تنظیمی که در اینجا نیاز است، قرار دادن ویژگی Table بر روی نام کلاس‌های بازیکن و مربی است. به این ترتیب حالت پیش فرض الف (TPH) اعمال نخواهد شد.

ج) Table per Concrete Type - TPC

در این حالت فقط دو جدول برای بازیکن و مربی تشکیل می‌شوند و جدولی برای شخص تشکیل نخواهد شد. خواص کلاس شخص، در هر دو جدول مربی و بازیکن به صورت جداگانه‌ای تکرار خواهد شد. تنظیم این مورد نیاز به استفاده از Fluent API دارد.

توضیحات بیشتر این موارد به همراه مثال، موکول خواهد شد به مباحث استفاده از Fluent API که برای تعریف تنظیمات پیشرفته نگاشت‌ها طراحی شده است. استفاده از متادیتا تنها قسمت کوچکی از توانایی‌های Fluent API را شامل می‌شود.

5) ConcurrencyCheck و Timestamp

هر دوی این ویژگی‌ها در فضای نام System.ComponentModel.DataAnnotations و اسمبلی به همین نام تعریف شده‌اند.

در EF Code first دو راه برای مدیریت مسایل همزمانی وجود دارد:

```
[ConcurrencyCheck]
public string Name { set; get; }

[Timestamp]
public byte[] RowVersion { set; get; }
```

زمانیکه از ویژگی ConcurrencyCheck استفاده می‌شود، تغییر خاصی در سمت بانک اطلاعاتی صورت نخواهد گرفت، اما در برنامه، کوئری‌های update و delete ایی که توسط EF صادر می‌شوند، اینبار اندکی متفاوت خواهند بود. برای مثال برنامه جاری را به نحو زیر تغییر دهید:

```
using System;
using System.Data.Entity;

namespace EF_Sample02
{
    class Program
    {
        static void Main(string[] args)
        {
            Database.SetInitializer(new Sample2DbInitializer());
            using (var db = new Sample2Context())
            {
                //update
                var user = db.Users.Find(1);
                user.Name = "User name 1";
                db.SaveChanges();
            }
        }
    }
}
```

متد Find بر اساس primary key عمل می‌کند. به این ترتیب، اول رکورد یافت شده و سپس نام آن تغییر کرده و در ادامه، اطلاعات ذخیره خواهند شد.

اکنون اگر توسط SQL Server Profiler کوئری update حاصل را بررسی کنیم، به نحو زیر خواهد بود:

```
exec sp_executesql N'update [dbo].[Users]
set [Name] = @0
where (([Id] = @1) and ([Name] = @2))
',N'@0 nvarchar(max) ,@1 int,@2 nvarchar(max) ',@0=N'User name 1',@1=1,@2=N'Vahid'
```

همانطور که ملاحظه می‌کنید، برای به روز رسانی فقط از primary key جهت یافتن رکورد استفاده نکرده، بلکه فیلد Name را نیز دخالت داده است. از این جهت که مطمئن شود در این بین، رکوردی که در حال به روز رسانی آن هستیم، توسط کاربر دیگری در شبکه تغییر نکرده باشد و اگر در این بین تغییری رخ داده باشد، یک استثناء صادر خواهد شد.
همین رفتار در مورد delete نیز وجود دارد:

```
//delete
var user = db.Users.Find(1);
db.Users.Remove(user);
db.SaveChanges();
```

که خروجی آن به صورت زیر است:

```
exec sp_executesql N'delete [dbo].[Users]
where (([Id] = @0) and ([Name] = @1))',N'@0 int,@1 nvarchar(max) ',@0=1,@1=N'Vahid'
```

در اینجا نیز به علت مزین بودن خاصیت Name به ویژگی ConcurrencyCheck، فقط همان رکوردی که یافت شده باید حذف شود و نه نمونه تغییر یافته آن توسط کاربری دیگر در شبکه.
البته در این مثال شاید این پروسه تنها چند میلی ثانیه به نظر برسد. اما در برنامه‌ای با رابط کاربری، شخصی ممکن است اطلاعات یک رکورد را در یک صفحه دریافت کرده و 5 دقیقه بعد بر روی دکمه save کلیک کند. در این بین ممکن است شخص دیگری در شبکه همین رکورد را تغییر داده باشد. بنابراین اطلاعاتی را که شخص مشاهده می‌کند، فاقد اعتبار شده‌اند.

ConcurrencyCheck را بر روی هر فیلدی می‌توان بکاربرد، اما ویژگی Timestamp کاربرد مشخص و محدودی دارد. باید به خاصیتی از نوع byte array اعمال شود (که نمونه‌ای از آن‌را در بالا در خاصیت public byte[] RowVersion مشاهده نمودید). علاوه بر آن، این ویژگی بر روی بانک اطلاعاتی نیز تاثیر دارد (نوع فیلد را در SQL Server تبدیل به timestamp می‌کند و نه از نوع varbinary مانند فیلد تصویر). SQL Server با این نوع فیلد به خوبی آشنا است و قابلیت مقدار دهی خودکار آن‌را دارد. بنابراین نیازی نیست در حین تشکیل اشیاء در برنامه، قید شود.
پس از آن، این فیلد مقدار دهی شده به صورت خودکار توسط بانک اطلاعاتی، در تمام updateها و deleteهای EF Code first حضور خواهد داشت:

```
exec sp_executesql N'delete [dbo].[Users]
where ((([Id] = @0) and ([Name] = @1)) and ([RowVersion] = @2))',N'@0 int,@1 nvarchar(max) ,
@2 binary(8)',@0=1,@1=N'Vahid',@2=0x00000000000007D1
```

از این جهت که اطمینان حاصل شود، واقعا مشغول به روز رسانی یا حذف رکوردی هستیم که در ابتدای عملیات از بانک اطلاعاتی دریافت کرده‌ایم. اگر در این بین RowVesrion تغییر کرده باشد، یعنی کاربر دیگری در شبکه این رکورد را تغییر داده و ما در حال حاضر مشغول به کار با رکوردی غیرمعتبر هستیم.
بنابراین استفاده از Timestamp را می‌توان به عنوان یکی از best practices طراحی برنامه‌های چند کاربره ASP.NET درنظر داشت.

6) NotMapped و DatabaseGenerated

این دو ویژگی نیز در فضای نام System.ComponentModel.DataAnnotations قرار دارند، اما در اسمبلی EntityFramework.dll تعریف شده‌اند.
به کمک ویژگی DatabaseGenerated، مشخص خواهیم کرد که این فیلد قرار است توسط بانک اطلاعاتی تولید شود. برای مثال خواصی از نوع public int Id به صورت خودکار به فیلدهایی از نوع identity که توسط بانک اطلاعاتی تولید می‌شوند، نگاشت خواهند شد و نیازی نیست تا به صورت صریح از ویژگی DatabaseGenerated جهت مزین سازی آن‌ها کمک گرفت. البته اگر

علاقمند نیستید که primary key شما از نوع identity باشد، می‌توانید از گزینه DatabaseGeneratedOption.None استفاده نمائید:

```
[DatabaseGenerated(DatabaseGeneratedOption.None)]
public int Id { set; get; }
```

DatabaseGeneratedOption در اینجا یک enum است که به نحو زیر تعریف شده است:

```
public enum DatabaseGeneratedOption
{
        None = 0,
        Identity = 1,
        Computed = 2
}
```

تا اینجا حالت‌های None و Identity آن، بحث شدند.

در SQL Server امکان تعریف فیلدهای محاسباتی و Computed با T-SQL نویسی نیز وجود دارد. این نوع فیلدها در هربار insert یا update یک رکورد، به صورت خودکار توسط بانک اطلاعاتی مقدار دهی می‌شوند. بنابراین اگر قرار است خاصیتی به این نوع فیلدها در SQL Server نگاشت شود، می‌توان از گزینه DatabaseGeneratedOption.Computed استفاده کرد.

یا اگر برای فیلدی در بانک اطلاعاتی default value تعریف کرده‌اید، مثلا برای فیلد date متد getdate توکار SQL Server را به عنوان پیش فرض درنظر گرفته‌اید و قرار هم نیست توسط برنامه مقدار دهی شود، باز هم می‌توان آن‌را از نوع DatabaseGeneratedOption.Computed تعریف کرد.

البته باید درنظر داشت که اگر خاصیت DateTime تعریف شده در اینجا به همین نحو بکاربرده شود، اگر مقداری برای آن در حین تعریف یک وهله جدید از کلاس User درکدهای برنامه درنظر گرفته نشود، یک مقدار پیش فرض حداقل به آن انتساب داده خواهد شد (چون value type است). بنابراین نیاز است این خاصیت را از نوع nullable تعریف کرد (public DateTime? AddDate).

همچنین اگر یک خاصیت محاسباتی در کلاسی به صورت ReadOnly تعریف شده است (توسط کدهای مثلا سی شارپ یا وی بی):

```
[NotMapped]
public string FullName
{
    get { return Name + " " + LastName; }
}
```

بدیهی است نیازی نیست تا آن‌را به یک فیلد بانک اطلاعاتی نگاشت کرد. این نوع خواص را با ویژگی NotMapped می‌توان مزین کرد.

همچنین باید دقت داشت در این حالت، از این نوع خواص دیگر نمی‌توان در کوئری‌های EF استفاده کرد. چون نهایتا این کوئری‌ها قرار هستند به عبارات SQL ترجمه شوند و چنین فیلدی در جدول بانک اطلاعاتی وجود ندارد. البته بدیهی است امکان تهیه کوئری LINQ to Objects (کوئری از اطلاعات درون حافظه) همیشه مهیا است و اهمیتی ندارد که این خاصیت درون بانک اطلاعاتی معادلی دارد یا خیر.

7) ComplexType

ComplexType یا Component mapping مربوط به حالتی است که شما یک سری خواص را در یک کلاس تعریف می‌کنید، اما قصد ندارید این‌ها واقعا تبدیل به یک جدول مجزا (به همراه کلید خارجی) در بانک اطلاعاتی شوند. می‌خواهید این خواص دقیقا در همان جدول اصلی کنار مابقی خواص قرار گیرند؛ اما در طرف کدهای ما به شکل یک کلاس مجزا تعریف و مدیریت شوند.

یک مثال:

کلاس زیر را به همراه ویژگی ComplexType به برنامه مطلب جاری اضافه نمائید:

```
using System.ComponentModel.DataAnnotations;

namespace EF_Sample02.Models
{
    [ComplexType]
    public class InterestComponent
    {
        [MaxLength(450, ErrorMessage = "حداکثر 450 حرف")]
        public string Interest1 { get; set; }

        [MaxLength(450, ErrorMessage = "حداکثر 450 حرف")]
        public string Interest2 { get; set; }
    }
}
```

سپس خاصیت زیر را نیز به کلاس User اضافه کنید:

```
public InterestComponent Interests { set; get; }
```

همانطور که ملاحظه می‌کنید کلاس InterestComponent فاقد Id است؛ بنابراین هدف از آن تعریف یک Entity نیست و قرار هم نیست در کلاس مشتق شده از DbContext تعریف شود. از آن صرفا جهت نظم بخشیدن به یک سری خاصیت مرتبط و هم‌خانواده استفاده شده است (مثلا آدرس یک، آدرس 2، تا آدرس 10 یک شخص، یا تلفن یک تلفن 2 یا موبایل 10 یک شخص).
اکنون اگر پروژه را اجرا نمائیم، ساختار جدول کاربر به نحو زیر تغییر خواهد کرد:

```
CREATE TABLE [dbo].[Users](
 ---...
 [Interests_Interest1] [nvarchar](450) NULL,
 [Interests_Interest2] [nvarchar](450) NULL,
 ---...
```

در اینجا خواص کلاس InterestComponent، داخل همان کلاس User تعریف شده‌اند و نه در یک جدول مجزا. تنها در سمت کدهای ما است که مدیریت آن‌ها منطقی‌تر شده‌اند.

یک نکته:
یکی از الگوهایی که حین تشکیل مدل‌های برنامه عموما مورد استفاده قرار می‌گیرد، null object pattern نام دارد. برای مثال:

```
namespace EF_Sample02.Models
{
    public class User
    {
        public InterestComponent Interests { set; get; }
        public User()
        {
            Interests = new InterestComponent();
        }
    }
}
```

در اینجا در سازنده کلاس User، به خاصیت Interests وهله‌ای از کلاس InterestComponent نسبت داده شده است. به این

ترتیب دیگر در کدهای برنامه مدام نیازی نخواهد بود تا بررسی شود که آیا Interests نال است یا خیر. همچنین استفاده از این الگو حین کار با یک ComplexType ضروری است؛ زیرا EF امکان ثبت رکورد جاری را در صورت نال بودن خاصیت Interests (صرفنظر از اینکه خواص آن مقدار دهی شده‌اند یا خیر) نخواهد داد.

8) ForeignKey

این ویژگی نیز در فضای نام System.ComponentModel.DataAnnotations قرار دارد، اما در اسمبلی EntityFramework.dll تعریف شده‌است.

اگر از قراردادهای پیش فرض نامگذاری کلیدهای خارجی در EF Code first خرسند نیستید، می‌توانید توسط ویژگی ForeignKey، نامگذاری مورد نظر خود را اعمال نمائید. باید دقت داشت که ویژگی ForeignKey را باید به یک Reference property اعمال کرد. همچنین در این حالت، کلید خارجی را با یک value type نیز می‌توان نمایش داد:

```
[ForeignKey("FK_User_Id")]
public virtual User User { set; get; }
public int FK_User_Id { set; get; }
```

در اینجا فیلد اضافی دوم FK_User_Id به جدول Project اضافه نخواهد شد (چون توسط ویژگی ForeignKey تعریف شده است و فقط یکبار تعریف می‌شود). اما در این حالت نیز وجود Reference property ضروری است.

9) InverseProperty

این ویژگی نیز در فضای نام System.ComponentModel.DataAnnotations قرار دارد، اما در اسمبلی EntityFramework.dll تعریف شده‌است.

از ویژگی InverseProperty برای تعریف روابط دو طرفه استفاده می‌شود.

برای مثال دو کلاس زیر را درنظر بگیرید:

```
public class Book
{
    public int ID {get; set;}
    public string Title {get; set;}

    [InverseProperty("Books")]
    public Author Author {get; set;}
}

public class Author
{
    public int ID {get; set;}
    public string Name {get; set;}

    [InverseProperty("Author")]
    public virtual ICollection<Book> Books {get; set;}
}
```

این دو کلاس همانند کلاس‌های User و Project فوق هستند. ذکر ویژگی InverseProperty برای مشخص سازی ارتباطات بین این دو غیرضروری است و قراردادهای توکار EF Code first یک چنین مواردی را به خوبی مدیریت می‌کنند.

اما اکنون مثال زیر را درنظر بگیرید:

```
public class Book
{
    public int ID {get; set;}
    public string Title {get; set;}

    public Author FirstAuthor {get; set;}
    public Author SecondAuthor {get; set;}
}

public class Author
{
```

```
    public int ID {get; set;}
    public string Name {get; set;}

    public virtual ICollection<Book> BooksAsFirstAuthor {get; set;}
    public virtual ICollection<Book> BooksAsSecondAuthor {get; set;}
}
```

این مثال ویژه‌ای است از کتابخانه‌ای که کتاب‌های آن، تنها توسط دو نویسنده نوشته شده‌اند. اگر برنامه را بر اساس این دو کلاس اجرا کنیم، EF Code first قادر نخواهد بود تشخیص دهد، روابط کدام به کدام هستند و در جدول Books چهار کلید خارجی را ایجاد می‌کند. برای مدیریت این مساله و تعین ابتدا و انتهای روابط می‌توان از ویژگی InverseProperty کمک گرفت:

```
public class Book
{
    public int ID {get; set;}
    public string Title {get; set;}

    [InverseProperty("BooksAsFirstAuthor")]
    public Author FirstAuthor {get; set;}
    [InverseProperty("BooksAsSecondAuthor")]
    public Author SecondAuthor {get; set;}
}

public class Author
{
    public int ID {get; set;}
    public string Name {get; set;}

    [InverseProperty("FirstAuthor")]
    public virtual ICollection<Book> BooksAsFirstAuthor {get; set;}
    [InverseProperty("SecondAuthor")]
    public virtual ICollection<Book> BooksAsSecondAuthor {get; set;}
}
```

اینبار اگر برنامه را اجرا کنیم، بین این دو جدول تنها دو رابطه تشکیل خواهد شد و نه چهار رابطه؛ چون EF اکنون می‌داند که ابتدا و انتهای روابط کجا است. همچنین ذکر ویژگی InverseProperty در یک سر رابطه کفایت می‌کند و نیازی به ذکر آن در طرف دوم نیست.

عنوان: EF Code First #4
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۱۷ ۱۳:۴۹:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Entity framework

آشنایی با Code first migrations

ویژگی Code first migrations برای اولین بار در EF 4.3 ارائه شد و هدف آن سهولت هماهنگ سازی کلاس‌های مدل برنامه با بانک اطلاعاتی است؛ به صورت خودکار یا با تنظیمات دقیق دستی.

همانطور که در قسمت‌های قبل نیز به آن اشاره شد، تا پیش از EF 4.3، پنج روال جهت آغاز به کار با بانک اطلاعاتی در EF code first وجود داشت و دارد:

1) در اولین بار اجرای برنامه، در صورتیکه بانک اطلاعاتی اشاره شده در رشته اتصالی وجود خارجی نداشته باشد، نسبت به ایجاد خودکار آن اقدام می‌گردد. اینکار پس از وهله سازی اولین DbContext و همچنین صدور یک کوئری به بانک اطلاعاتی انجام خواهد شد.

2) DropCreateDatabaseAlways : همواره پس از شروع برنامه، ابتدا بانک اطلاعاتی را drop کرده و سپس نمونه جدیدی را ایجاد می‌کند.

3) DropCreateDatabaseIfModelChanges : اگر EF Code first تشخیص دهد که تعاریف مدل‌های شما با بانک اطلاعاتی مشخص شده توسط رشته اتصالی، هماهنگ نیست، آن‌را drop کرده و نمونه جدیدی را تولید می‌کند.

4) با مقدار دهی پارامتر متد System.Data.Entity.Database.SetInitializer به نال، می‌توان فرآیند آغاز خودکار بانک اطلاعاتی را غیرفعال کرد. در این حالت شخص می‌تواند تغییرات انجام شده در کلاس‌های مدل برنامه را به صورت دستی به بانک اطلاعاتی اعمال کند.

5) می‌توان با پیاده سازی اینترفیس IDatabaseInitializer، یک آغاز کننده بانک اطلاعاتی سفارشی را نیز تولید کرد.

اکثر این روش‌ها در حین توسعه یک برنامه یا خصوصا جهت سهولت انجام آزمون‌های خودکار بسیار مناسب هستند، اما به درد محیط کاری نمی‌خورند؛ زیرا drop یک بانک اطلاعاتی به معنای از دست دادن تمام اطلاعات ثبت شده در آن است. برای رفع این مشکل مهم، مفهومی به نام «Migrations» در EF 4.3 ارائه شده است تا بتوان بانک اطلاعاتی را بدون تخریب آن، بر اساس اطلاعات تغییر کرده‌ی کلاس‌های مدل برنامه، تغییر داد. البته بدیهی است زمانیکه توسط NuGet نسبت به دریافت و نصب EF اقدام می‌شود، همواره آخرین نگارش پایدار که حاوی اطلاعات و فایل‌های مورد نیاز جهت کار با «Migrations» است را نیز دریافت خواهیم کرد.

تنظیمات ابتدایی Code first migrations

در اینجا قصد داریم همان مثال قسمت قبل را ادامه دهیم. در آن مثال از یک نمونه سفارشی سازی شده DropCreateDatabaseAlways استفاده شد.

نیاز است از منوی Tools در ویژوال استودیو، گزینه Library package manager آن، گزینه package manager console را انتخاب کرد تا کنسول پاورشل NuGet ظاهر شود.

اطلاعات مرتبط با پاورشل EF، به صورت خودکار توسط NuGet نصب می‌شود. برای مثال جهت مشاهده آن‌ها به مسیر packages\EntityFramework.4.3.1\tools در کنار پوشه پروژه خود مراجعه نمائید.

در ادامه در پایین صفحه، زمانیکه کنسول پاورشل NuGet ظاهر می‌شود، ابتدا باید دقت داشت که قرار است فرامین را بر روی چه پروژه‌ای اجرا کنیم. برای مثال اگر تعاریف DbContext را به یک اسمبلی و پروژه class library مجزا انتقال داده‌اید، گزینه Default project را در این قسمت باید به این پروژه مجزا، تغییر دهید.

سپس در خط فرمان پاور شل، دستور enable-migrations را وارد کرده و دکمه enter را فشار دهید.

پس از اجرای این دستور، یک سری اتفاقات رخ خواهد داد:

الف) پوشه‌ای به نام Migrations به پروژه پیش فرض مشخص شده در کنسول پاورشل، اضافه می‌شود.
ب) دو کلاس جدید نیز در آن پوشه تعریف خواهند شد به نام‌های Configuration.cs و یک نام خودکار مانند number_InitialCreate.cs
ج) در کنسول پاور شل، پیغام زیر ظاهر می‌گردد:

```
Detected database created with a database initializer. Scaffolded migration
'201205050805256_InitialCreate'
corresponding to current database schema. To use an automatic migration instead, delete the Migrations
folder and re-run Enable-Migrations specifying the -EnableAutomaticMigrations parameter.
```

با توجه به اینکه در مثال قسمت سوم، از آغاز کننده سفارشی سازی شده DropCreateDatabaseAlways استفاده شده بود، اطلاعات آن در جدول سیستمی dbo.__MigrationHistory در بانک اطلاعاتی برنامه موجود است (تصویری از آن‌را در قسمت اول این سری مشاهده کردید). سپس با توجه به ساختار بانک اطلاعاتی جاری، دو کلاس خودکار زیر را ایجاد کرده است:

```
namespace EF_Sample02.Migrations
{
    using System;
    using System.Data.Entity;
    using System.Data.Entity.Migrations;
    using System.Linq;

    internal sealed class Configuration : DbMigrationsConfiguration<EF_Sample02.Sample2Context>
    {
        public Configuration()
        {
            AutomaticMigrationsEnabled = false;
        }

        protected override void Seed(EF_Sample02.Sample2Context context)
        {
            //  This method will be called after migrating to the latest version.

            //  You can use the DbSet<T>.AddOrUpdate() helper extension method
            //  to avoid creating duplicate seed data. E.g.
            //
            //    context.People.AddOrUpdate(
            //      p => p.FullName,
            //      new Person { FullName = "Andrew Peters" },
            //      new Person { FullName = "Brice Lambson" },
            //      new Person { FullName = "Rowan Miller" }
            //    );
            //
        }
    }
}
```

```
namespace EF_Sample02.Migrations
{
    using System.Data.Entity.Migrations;

    public partial class InitialCreate : DbMigration
    {
        public override void Up()
        {
            CreateTable(
                "Users",
                c => new
                    {
                        Id = c.Int(nullable: false, identity: true),
                        Name = c.String(),
                        LastName = c.String(),
                        Email = c.String(),
                        Description = c.String(),
                        Photo = c.Binary(),
                        RowVersion = c.Binary(nullable: false, fixedLength: true, timestamp: true,
storeType: "rowversion"),
                        Interests_Interest1 = c.String(maxLength: 450),
```

```
                        Interests_Interest2 = c.String(maxLength: 450),
                        AddDate = c.DateTime(nullable: false),
                    })
                .PrimaryKey(t => t.Id);

            CreateTable(
                "Projects",
                c => new
                    {
                        Id = c.Int(nullable: false, identity: true),
                        Title = c.String(maxLength: 50),
                        Description = c.String(),
                        RowVesrion = c.Binary(nullable: false, fixedLength: true, timestamp: true,
storeType: "rowversion"),
                        AddDate = c.DateTime(nullable: false),
                        AdminUser_Id = c.Int(),
                    })
                .PrimaryKey(t => t.Id)
                .ForeignKey("Users", t => t.AdminUser_Id)
                .Index(t => t.AdminUser_Id);

        }

        public override void Down()
        {
            DropIndex("Projects", new[] { "AdminUser_Id" });
            DropForeignKey("Projects", "AdminUser_Id", "Users");
            DropTable("Projects");
            DropTable("Users");
        }
    }
}
```

در این کلاس خودکار، نحوه ایجاد جداول بانک اطلاعاتی تعریف شده‌اند. در متد تحریف شده Up، کار ایجاد بانک اطلاعاتی و در متد تحریف شده Down، دستورات حذف جداول و قیود ذکر شده‌اند.

به علاوه اینبار متد Seed را در کلاس مشتق شده از DbMigrationsConfiguration، می‌توان تحریف و مقدار دهی کرد.

علاوه بر این‌ها جدول سیستمی dbo.__MigrationHistory نیز با اطلاعات جاری مقدار دهی می‌گردد.

فعال سازی گزینه‌های مهاجرت خودکار

برای استفاده از این کلاس‌ها، ابتدا به فایل Configuration.cs مراجعه کرده و خاصیت AutomaticMigrationsEnabled را true کنید:

```
internal sealed class Configuration : DbMigrationsConfiguration<EF_Sample02.Sample2Context>
{
        public Configuration()
        {
            AutomaticMigrationsEnabled = true;
        }
```

پس از آن EF به صورت خودکار کار استفاده و مدیریت «Migrations» را عهده‌دار خواهد شد. البته برای این منظور باید نوع آغاز کننده بانک اطلاعاتی را از DropCreateDatabaseAlways قبلی به نمونه جدید MigrateDatabaseToLatestVersion نیز تغییر دهیم:

```
//Database.SetInitializer(new Sample2DbInitializer());
Database.SetInitializer(new MigrateDatabaseToLatestVersion<Sample2Context,
Migrations.Configuration>());
```

یک نکته:

کلاس Migrations.Configuration که باید در حین وهله سازی از MigrateDatabaseToLatestVersion قید شود (همانند کدهای فوق)، از نوع internal sealed معرفی شده است. بنابراین اگر این کلاس را در یک اسمبلی جداگانه قرار داده‌اید، نیاز است فایل را ویرایش کرده و internal sealed آن‌را به public تغییر دهید.

روش دیگر معرفی کلاس‌های Context و Migrations.Configuration، حذف متد Database.SetInitializer و استفاده از فایل app.config یا web.config است به نحو زیر (در اینجا حرف ` اصطلاحا back tick نام دارد. فشردن دکمه ~ در حین تایپ انگلیسی):

```
<entityFramework>
    <contexts>
      <context type="EF_Sample02.Sample2Context, EF_Sample02">
        <databaseInitializer
           type="System.Data.Entity.MigrateDatabaseToLatestVersion`2[[EF_Sample02.Sample2Context,
EF_Sample02],
                [EF_Sample02.Migrations.Configuration, EF_Sample02]], EntityFramework"
        />
      </context>
    </contexts>
  </entityFramework>
```

آزمودن ویژگی مهاجرت خودکار

اکنون برای آزمایش این موارد، یک خاصیت دلخواه را به کلاس Project به نام public string SomeProp اضافه کنید. سپس برنامه را اجرا نمائید.
در ادامه به بانک اطلاعاتی مراجعه کرده و فیلدهای جدول Projects را بررسی کنید:

```
CREATE TABLE [dbo].[Projects](
---...
 [SomeProp] [nvarchar](max) NULL,
---...
```

بله. اینبار فیلد SomeProp بدون از دست رفتن اطلاعات و drop بانک اطلاعاتی، به جدول پروژه‌ها اضافه شده است.

عکس العمل ویژگی مهاجرت خودکار در مقابل از دست رفتن اطلاعات

در ادامه، خاصیت public string SomeProp را که در قسمت قبل به کلاس پروژه اضافه کردیم، حذف کنید. اکنون مجددا برنامه را اجرا نمائید. برنامه بلافاصله با استثنای زیر متوقف خواهد شد:

```
Automatic migration was not applied because it would result in data loss.
```

از آنجائیکه حذف یک خاصیت مساوی است با حذف یک ستون در جدول بانک اطلاعاتی، امکان از دست رفتن اطلاعات در این بین بسیار زیاد است. بنابراین ویژگی مهاجرت خودکار دیگر اعمال نخواهد شد و این مورد به نوعی یک محافظت خودکار است که درنظر گرفته شده است.
البته در EF Code first این مساله را نیز می‌توان کنترل نمود. به کلاس Configuration اضافه شده توسط پاورشل مراجعه کرده و خاصیت AutomaticMigrationDataLossAllowed را به true تنظیم کنید:

```
internal sealed class Configuration : DbMigrationsConfiguration<EF_Sample02.Sample2Context>
{
        public Configuration()
        {
            this.AutomaticMigrationsEnabled = true;
            this.AutomaticMigrationDataLossAllowed = true;
        }
```

این تغییر به این معنا است که خودمان صریحا مجوز حذف یک ستون و اطلاعات مرتبط به آن‌را صادر کرده‌ایم.
پس از این تغییر، مجددا برنامه را اجرا کنید. ستون SomeProp به صورت خودکار حذف خواهد شد، اما اطلاعات رکوردهای موجود تغییری نخواهند کرد.

استفاده از Code first migrations بر روی یک بانک اطلاعاتی موجود

تفاوت یک دیتابیس موجود با بانک اطلاعاتی تولید شده توسط EF Code first در نبود جدول سیستمی dbo.__MigrationHistory است.
به این ترتیب زمانیکه فرمان enable-migrations را در یک پروژه EF code first متصل به بانک اطلاعاتی قدیمی موجود اجرا می‌کنیم، پوشه Migration در آن ایجاد خواهد شد اما تنها حاوی فایل Configuration.cs است و نه فایلی شبیه به number_InitialCreate.cs .
بنابراین نیاز است به صورت صریح به EF اعلام کنیم که نیاز است تا جدول سیستمی dbo.__MigrationHistory و فایل number_InitialCreate.cs را نیز تولید کند. برای این منظور کافی است دستور زیر را در خط فرمان پاورشل NuGet پس از فراخوانی enable-migrations اولیه، اجرا کنیم:

```
add-migration Initial -IgnoreChanges
```

با بکارگیری پارامتر IgnoreChanges، متد Up در فایل number_InitialCreate.cs تولید نخواهد شد. به این ترتیب نگران نخواهیم بود که در اولین بار اجرای برنامه، تعاریف دیتابیس موجود ممکن است اندکی تغییر کند.
سپس دستور زیر را جهت به روز رسانی جدول سیستمی dbo.__MigrationHistory اجرا کنید:

```
update-database
```

پس از آن جهت سوئیچ به مهاجرت خودکار، خاصیت AutomaticMigrationsEnabled = true را در فایل Configuration.cs همانند قبل مقدار دهی کنید.

مشاهده دستوارت SQL به روز رسانی بانک اطلاعاتی

اگر علاقمند هستید که دستورات T-SQL به روز رسانی بانک اطلاعاتی را نیز مشاهده کنید، دستور Update-Database را با پارامتر Verbose آغاز نمائید:

```
Update-Database -Verbose
```

و اگر تنها نیاز به مشاهده اسکریپت تولیدی بدون اجرای آن‌ها بر روی بانک اطلاعاتی مدنظر است، از پارامتر Script باید استفاده کرد:

```
update-database -Script
```

نکته‌ای در مورد جدول سیستمی dbo.__MigrationHistory

تنها دلیلی که این جدول در SQL Server البته (ونه برای مثال در SQL Server CE) به صورت سیستمی معرفی می‌شود این است که «جلوی چشم نباشد»! به این ترتیب در SQL Server management studio در بین سایر جداول معمولی بانک اطلاعاتی قرار نمی‌گیرد. اما برای EF تفاوتی نمی‌کند که این جدول سیستمی است یا خیر.
همین سیستمی بودن آن ممکن است بر اساس سطح دسترسی کاربر اتصالی به بانک اطلاعاتی مساله ساز شود. برای نمونه ممکن است schema کاربر متصل dbo نباشد. همینجا است که کار به روز رسانی این جدول متوقف خواهد شد.
بنابراین اگر قصد داشتید خواص سیستمی آن‌را لغو کنید، تنها کافی است دستورات T-SQL زیر را در SQL Server اجرا نمائید:

```
SELECT * INTO [TempMigrationHistory]
FROM [__MigrationHistory]
DROP TABLE [__MigrationHistory]
EXEC sp_rename [TempMigrationHistory], [__MigrationHistory]
```

ساده سازی پروسه مهاجرت خودکار

کل پروسه‌ای را که در این قسمت مشاهده کردید، به صورت ذیل نیز می‌توان خلاصه کرد:

```
using System;
using System.Data.Entity;
using System.Data.Entity.Migrations;
using System.Data.Entity.Migrations.Infrastructure;
using System.IO;

namespace EF_Sample02
{
    public class Configuration<T> : DbMigrationsConfiguration<T> where T : DbContext
    {
        public Configuration()
        {
            AutomaticMigrationsEnabled = true;
            AutomaticMigrationDataLossAllowed = true;
        }
    }

    public class SimpleDbMigrations
    {
        public static void UpdateDatabaseSchema<T>(string SQLScriptPath = "script.sql") where T :
DbContext
        {
            var configuration = new Configuration<T>();
            var dbMigrator = new DbMigrator(configuration);
            saveToFile(SQLScriptPath, dbMigrator);
            dbMigrator.Update();
        }

        private static void saveToFile(string SQLScriptPath, DbMigrator dbMigrator)
        {
            if (string.IsNullOrWhiteSpace(SQLScriptPath)) return;

            var scriptor = new MigratorScriptingDecorator(dbMigrator);
            var script = scriptor.ScriptUpdate(sourceMigration: null, targetMigration: null);
            File.WriteAllText(SQLScriptPath, script);
            Console.WriteLine(script);
        }
```

```
    }
}
```

سپس برای استفاده از آن خواهیم داشت:

```
SimpleDbMigrations.UpdateDatabaseSchema<Sample2Context>();
```

در این کلاس ذخیره سازی اسکریپت تولیدی جهت به روز رسانی بانک اطلاعاتی جاری در یک فایل نیز درنظر گرفته شده است.

تا اینجا مهاجرت خودکار را بررسی کردیم. در قسمت بعدی Code-Based Migrations را ادامه خواهیم داد.

عنوان: EF Code First #5
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۱۸ ۰۹:۰۵:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Entity framework

در قسمت قبل خاصیت AutomaticMigrationsEnabled را در کلاس Configuration به true تنظیم کردیم. به این ترتیب، عملیات ساده شده، اما یک سری از قابلیت‌های ردیابی تغییرات را از دست خواهیم داد و این عملیات، صرفا یک عملیات رو به جلو خواهد بود.
اگر AutomaticMigrationsEnabled را مجددا به false تنظیم کنیم و هربار به کمک دستوارت Add-Migration و Update-Database تغییرات مدل‌ها را به بانک اطلاعاتی اعمال نمائیم، علاوه بر تشکیل تاریخچه این تغییرات در برنامه، امکان بازگشت به عقب و لغو تغییرات صورت گرفته نیز مهیا می‌گردد.

هدف قرار دادن مرحله‌ای خاص یا لغو آن

به همان پروژه قسمت قبل مراجعه نمائید. در کلاس Configuration آن، خاصیت AutomaticMigrationsEnabled را به false تنظیم کنید. سپس یک خاصیت جدید را به کلاس Project اضافه نموده و برنامه را اجرا نمائید. بلافاصله خطای زیر را دریافت خواهیم کرد:

```
Unable to update database to match the current model because there are pending changes and
automatic migration is disabled. Either write the pending model changes to a code-based migration
or enable automatic migration. Set DbMigrationsConfiguration.AutomaticMigrationsEnabled to true
to enable automatic migration.
```

EF تشخیص داده است که کلاس مدل برنامه، با بانک اطلاعاتی تطابق ندارد و همچنین ویژگی مهاجرت خودکار نیز فعال نیست. بنابراین اعمال code-based migration را توصیه کرده است.
برای این منظور به کنسول پاورشل NuGet مراجعه نمائید (منوی Tools در ویژوال استودیو، گزینه Library package manager آن و سپس انتخاب گزینه package manager console). در ادامه فرمان add-m را نوشته و دکمه tab را فشار دهید. یک منوی Auto Complete ظاهر خواهد شد که از آن می‌توان فرمان add-migration را انتخاب نمود. در اینجا یک نام را هم نیاز است وارد کرد؛ برای مثال:

```
Add-Migration AddSomeProp2ToProject
```

به این ترتیب کلاس زیر را به صورت خودکار تولید خواهد کرد:

```
namespace EF_Sample02.Migrations
{
    using System.Data.Entity.Migrations;

    public partial class AddSomeProp2ToProject : DbMigration
    {
        public override void Up()
        {
            AddColumn("Projects", "SomeProp", c => c.String());
            AddColumn("Projects", "SomeProp2", c => c.String());
        }

        public override void Down()
```

```
        {
            DropColumn("Projects", "SomeProp2");
            DropColumn("Projects", "SomeProp");
        }
    }
}
```

مدل‌های برنامه را با بانک اطلاعاتی تطابق داده و دریافته است که هنوز دو خاصیت در اینجا به بانک اطلاعاتی اضافه نشده‌اند.
از متد Up برای اعمال تغییرات و از متد Down برای بازگشت به قبل استفاده می‌گردد. نام فایل این کلاس هم طبق معمول چیزی است شبیه به timeStamp_AddSomeProp2ToProject.cs .

در ادامه نیاز است این تغییرات به بانک اطلاعاتی اعمال شوند. به همین منظور دستور زیر را در کنسول پاورشل وارد نمائید:

```
Update-Database -Verbose
```

پارامتر Verbose آن سبب خواهد شد تا جزئیات عملیات به صورت مفصل گزارش داده شود که شامل دستورات ALTER TABLE نیز هست:

```
Using NuGet project 'EF_Sample02'.
Using StartUp project 'EF_Sample02'.
Target database is: 'testdb2012' (DataSource: (local), Provider: System.Data.SqlClient, Origin:
Configuration).
Applying explicit migrations: [201205061835024_AddSomeProp2ToProject].
Applying explicit migration: 201205061835024_AddSomeProp2ToProject.
ALTER TABLE [Projects] ADD [SomeProp] [nvarchar](max)
ALTER TABLE [Projects] ADD [SomeProp2] [nvarchar](max)
[Inserting migration history record]
```

اکنون مجددا یک خاصیت دیگر را مثلا به نام public string SomeProp3، به کلاس Project اضافه نمائید.
سپس همین روال باید مجددا تکرار شود. دستورات زیر را در کنسول پاورشل NuGet اجرا نمائید:

```
Add-Migration AddSomeProp3ToProject
Update-Database -Verbose
```

اینبار نیز یک کلاس جدید به نام AddSomeProp3ToProject به پروژه اضافه خواهد شد و سپس بر اساس آن، امکان به روز رسانی بانک اطلاعاتی میسر می‌گردد.

در ادامه برای مثال به این نتیجه رسیده‌ایم که نیازی به خاصیت public string SomeProp3 اضافه شده، نبوده است. روش متداول، باز هم مانند سابق است. ابتدا خاصیت را از کلاس Project حذف خواهیم کرد و سپس دو دستور Add-Migration و Update-Database را اجرا خواهیم نمود.
اما با توجه به اینکه مهاجرت خودکار را غیرفعال کرده‌ایم و هربار با فراخوانی دستور Add-Migration یک کلاس جدید، با متدهای Up و Down به پروژه، جهت نگهداری سوابق عملیات اضافه می‌شوند، می‌توان دستور Update-Database را جهت فراخوانی متد Down صرفا یک مرحله موجود نیز فراخوانی نمود.

نکته:

اگر علاقمند باشید که راهنمای مفصل پارامترهای دستور Update-Database را مشاهده کنید، تنها کافی است دستور زیر را در کنسول پاورشل اجرا نمائید:

```
get-help update-database -detailed
```

به عنوان نمونه اگر در حین فراخوانی دستور Update-Database احتمال از دست رفتن اطلاعات باشد، عملیات متوقف می‌شود. برای وادار کردن پروسه به انجام تغییرات بر روی بانک اطلاعاتی می‌توان از پارامتر Force در اینجا استفاده کرد.

در ادامه برای اینکه دستور Update-Database تنها یک مرحله مشخص را که سابقه آن در برنامه موجود است، هدف قرار دهد، باید از پارامتر TargetMigration به همراه نام کلاس مرتبط استفاده کرد:

```
Update-Database -TargetMigration:"AddSomeProp2ToProject" -Verbose
```

اگر دقت کرده باشید در اینجا AddSomeProp **2** ToProject بجای AddSomeProp **3** ToProject بکارگرفته شده است. اگر یک مرحله قبل را هدف قرار دهیم، متد Down را اجرا خواهد کرد:

```
Using NuGet project 'EF_Sample02'.
Using StartUp project 'EF_Sample02'.
Target database is: 'testdb2012' (DataSource: (local), Provider: System.Data.SqlClient, Origin:
Configuration).
Reverting migrations: [201205061845485_AddSomeProp3ToProject].
Reverting explicit migration: 201205061845485_AddSomeProp3ToProject.
DECLARE @var0 nvarchar(128)
SELECT @var0 = name
FROM sys.default_constraints
WHERE parent_object_id = object_id(N'Projects')
AND col_name(parent_object_id, parent_column_id) = 'SomeProp3';
IF @var0 IS NOT NULL
    EXECUTE('ALTER TABLE [Projects] DROP CONSTRAINT ' + @var0)
ALTER TABLE [Projects] DROP COLUMN [SomeProp3]
[Deleting migration history record]
```

همانطور که ملاحظه می‌کنید در اینجا عملیات حذف ستون SomeProp3 انجام شده است. البته این خاصیت به صورت خودکار از کدهای برنامه (کلاس Project در این مثال) حذف نمی‌شود و فرض بر این است که پیشتر اینکار را انجام داده‌اید.

سفارشی سازی کلاس‌های مهاجرت

تمام کلاس‌های خودکار مهاجرت تولید شده توسط پاورشل، از کلاس DbMigration ارث بری می‌کنند. در این کلاس امکانات قابل توجهی مانند AddColumn، AddForeignKey، AddPrimaryKey، AlterColumn، CreateIndex و امثال آن وجود دارند که در تمام کلاس‌های مشتق شده از آن، قابل استفاده هستند. حتی متد Sql نیز در آن پیش بینی شده است که در صورت نیاز به اجرای دستوارت خام SQL، می‌توان از آن استفاده کرد.
برای مثال فرض کنید مجددا همان خاصیت public string SomeProp3 را به کلاس Project اضافه کرده‌ایم. اما اینبار نیاز است حین تشکیل این فیلد در بانک اطلاعاتی، یک مقدار پیش فرض نیز برای آن درنظر گرفته شود که در صورت نال بودن مقدار خاصیت آن در برنامه، به صورت خودکار توسط بانک اطلاعاتی مقدار دهی گردد:

```
namespace EF_Sample02.Migrations
```

```
{
    using System.Data.Entity.Migrations;

    public partial class AddSomeProp3ToProject : DbMigration
    {
        public override void Up()
        {
            AddColumn("Projects", "SomeProp3", c => c.String(defaultValue: "some data"));
            Sql("Update Projects set SomeProp3=N'some data'");
        }

        public override void Down()
        {
            DropColumn("Projects", "SomeProp3");
        }
    }
}
```

متد String در اینجا چنین امضایی دارد:

```
public ColumnModel String(bool? nullable = null, int? maxLength = null, bool? fixedLength = null,
bool? isMaxLength = null, bool? unicode = null, string defaultValue = null, string defaultValueSql =
null,
string name = null, string storeType = null)
```

که برای نمونه در اینجا پارامتر defaultValue آن‌را در کلاس AddSomeProp3ToProject مقدار دهی کرده‌ایم.
برای اعمال این تغییرات تنها کافی است دستور Update-Database -Verbose اجرا گردد. اینبار خروجی SQL اجرا شده آن به نحو زیر است که شامل مقدار پیش فرض نیز شده است:

```
ALTER TABLE [Projects] ADD [SomeProp3] [nvarchar](max) DEFAULT 'some data'
```

تعیین مقدار پیش فرض، زمانیکه یک فیلد not null تعریف شده‌است نیز می‌تواند مفید باشد. همچنین در اینجا امکان اجرای دستورات مستقیم SQL نیز وجود دارد که نمونه‌ای از آن‌را در متد Up فوق مشاهده می‌کنید.

افزودن رکوردهای پیش فرض در حین به روز رسانی بانک اطلاعاتی

در قسمت‌های قبل با متد Seed که به همراه آغاز کننده‌های بانک اطلاعاتی EF ارائه شده‌اند، جهت افزودن رکوردهای اولیه و پیش فرض به بانک اطلاعاتی آشنا شدید. در اینجا نیز با تحریف متد Seed در کلاس Configuration، چنین امری میسر است:

```
namespace EF_Sample02.Migrations
{
    using System;
    using System.Data.Entity.Migrations;

    internal sealed class Configuration : DbMigrationsConfiguration<EF_Sample02.Sample2Context>
    {
        public Configuration()
        {
            this.AutomaticMigrationsEnabled = false;
            this.AutomaticMigrationDataLossAllowed = true;
        }

        protected override void Seed(EF_Sample02.Sample2Context context)
        {
            context.Users.AddOrUpdate(
                 a => a.Name,
```

```
                new Models.User { Name = "Vahid", AddDate = DateTime.Now },
                new Models.User { Name = "Test", AddDate = DateTime.Now });
        }
    }
}
```

متد AddOrUpdate در EF 4.3 اضافه شده است. این متد ابتدا بررسی می‌کند که آیا رکورد مورد نظر در بانک اطلاعاتی وجود دارد یا خیر. اگر خیر، آن‌را اضافه خواهد کرد در غیراینصورت، نمونه موجود را به روز رسانی می‌کند. اولین پارامتر آن، identifierExpression نام دارد. توسط آن مشخص می‌شود که بر اساس چه خاصیتی باید در مورد update یا add تصمیم‌گیری شود. دراینجا اگر نیاز به ذکر بیش از یک خاصیت وجود داشت، از anonymously type object می‌توان کمک گرفت new { p.Name, p.LastName } .

تولید اسکریپت به روز رسانی بانک اطلاعاتی

بهترین کار و امن‌ترین روش حین انجام این نوع به روز رسانی‌ها، تهیه اسکریپت SQL فرامینی است که باید بر روی بانک اطلاعاتی اجرا شوند. سپس می‌توان این دستورات و اسکریپت نهایی را دستی هم اجرا کرد (که روش متداول‌تری است در محیط کاری).
برای اینکار تنها کافی است دستور زیر را در کنسول پاورشل اجرا نمائیم:

```
Update-Database -Verbose -Script
```

پس از اجرای این دستور، یک فایل اسکریپت با پسوند sql تولید شده و بلافاصله در ویژوال استودیو جهت مرور نیز گشوده خواهد شد. برای نمونه محتوای آن برای افزودن خاصیت جدید SomeProp5 به صورت زیر است:

```
ALTER TABLE [Projects] ADD [SomeProp5] [nvarchar](max)
INSERT INTO [__MigrationHistory] ([MigrationId], [CreatedOn], [Model], [ProductVersion]) VALUES
('201205060852004_AutomaticMigration', '2012-05-06T08:52:00.937Z', 0x1F8B0800000........... '4.3.1')
```

همانطور که ملاحظه می‌کنید، در یک مرحله، جدول پروژه‌ها را به روز خواهد کرد و در مرحله بعد، سابقه آن‌را در جدول __MigrationHistory ثبت می‌کند.

یک نکته:
اگر دستور فوق را بر روی برنامه‌ای که با بانک اطلاعاتی هماهنگ است اجرا کنیم، خروجی را مشاهده نخواهیم کرد. برای این منظور می‌توان مرحله خاصی را توسط پارامتر SourceMigration هدف گیری کرد:

```
Update-Database -Verbose -Script -SourceMigration:"stepName"
```

استفاده از DB Migrations در عمل

البته این یک روش پیشنهادی و امن است:
الف) در ابتدای اجرا برنامه، پارامتر ورودی متد System.Data.Entity.Database.SetInitializer را به نال تنظیم کنید تا برنامه تغییری را بر روی بانک اطلاعاتی اعمال نکند.
ب) توسط دستور enable-migrations، فایل‌های اولیه DB Migration را ایجاد کنید. پیش فرض‌های آن را نیز تغییر ندهید.

ج) هر بار که کلاس‌های مدل برنامه تغییر کردند و پس از آن نیاز به به روز رسانی ساختار بانک اطلاعاتی وجود داشت دو دستور زیر را اجرا کنید:

```
Add-Migration AddSomePropToProject
Update-Database -Verbose -Script
```

به این ترتیب سابقه تغییرات در برنامه نگهداری شده و همچنین بدون اجرای دستورات بر روی بانک اطلاعاتی، اسکریپت نهایی اعمال تغییرات تولید می‌گردد.

د) اسکریپت تولید شده را بررسی کرده و پس از تائید و افزودن به سورس کنترل، به صورت دستی بر روی بانک اطلاعاتی اجرا کنید (مثلا توسط management studio).

عنوان: EF Code First #6
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۱۹ ۰۰:۱۳:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Entity framework

ادامه بررسی Fluent API جهت تعریف نگاشت کلاس‌ها به بانک اطلاعاتی

در قسمت‌های قبل با استفاده از متادیتا و data annotations جهت بررسی نحوه نگاشت اطلاعات کلاس‌ها به جداول بانک اطلاعاتی آشنا شدیم. اما این موارد تنها قسمتی از توانایی‌های Fluent API مهیا در EF Code first را ارائه می‌دهند. یکی از دلایل آن هم به محدود بودن توانایی‌های ذاتی Attributes بر می‌گردد. برای مثال حین کار با Attributes امکان استفاده از متغیرها یا lambda expressions و امثال آن وجود ندارد. به علاوه شاید عده‌ای علاقمند نباشند تا کلاس‌های خود را با data annotations شلوغ کنند.

در قسمت دوم این سری، مروری مقدماتی داشتیم بر Fluent API. در آنجا ذکر شد که امکان تعریف نگاشت‌ها به کمک توانایی‌های Fluent API به دو روش زیر میسر است:
الف) می‌توان از متد protected override void OnModelCreating در کلاس مشتق شده از DbContext کار را شروع کرد.
ب) و یا اگر بخواهیم کلاس Context برنامه را شلوغ نکنیم بهتر است به ازای هر کلاس مدل برنامه، یک کلاس mapping مشتق شده از EntityTypeConfiguration را تعریف نمائیم. سپس می‌توان این کلاس‌ها را در متد OnModelCreating یاد شده، توسط متد modelBuilder.Configurations.Add جهت استفاده و اعمال، معرفی کرد.

کلاس‌های مدلی را که در این قسمت بررسی خواهیم کرد، همان کلاس‌های User و Project قسمت سوم هستند و هدف این قسمت بیشتر تطابق Fluent API با اطلاعات ارائه شده در قسمت سوم است؛ برای مثال در اینجا چگونه باید از خاصیتی صرفنظر کرد، مسایل همزمانی را اعمال نمود و امثال آن.
بنابراین یک پروژه جدید کنسول را آغاز نمائید. سپس با کمک NuGet ارجاعات لازم را به اسمبلی‌های EF اضافه نمائید.
در پوشه Models این پروژه، سه کلاس تکمیل شده زیر، از قسمت سوم وجود دارند:

```
using System;
using System.Collections.Generic;

namespace EF_Sample03.Models
{
    public class User
    {
        public int Id { set; get; }
        public DateTime AddDate { set; get; }
        public string Name { set; get; }
        public string LastName { set; get; }

        public string FullName
        {
            get { return Name + " " + LastName; }
        }

        public string Email { set; get; }
        public string Description { set; get; }
        public byte[] Photo { set; get; }
        public IList<Project> Projects { set; get; }
        public byte[] RowVersion { set; get; }
        public InterestComponent Interests { set; get; }

        public User()
        {
            Interests = new InterestComponent();
        }
    }
}
```

```
using System;

namespace EF_Sample03.Models
{
    public class Project
    {
        public int Id { set; get; }
        public DateTime AddDate { set; get; }
        public string Title { set; get; }
        public string Description { set; get; }
        public virtual User User { set; get; }
        public byte[] RowVesrion { set; get; }
    }
}
```

```
namespace EF_Sample03.Models
{
    public class InterestComponent
    {
        public string Interest1 { get; set; }
        public string Interest2 { get; set; }
    }
}
```

سپس یک پوشه جدید به نام Mappings را به پروژه اضافه نمائید. به ازای هر کلاس فوق، یک کلاس جدید را جهت تعاریف اطلاعات نگاشت‌ها به کمک Fluent API اضافه خواهیم کرد:

```
using System.Data.Entity.ModelConfiguration;
using EF_Sample03.Models;

namespace EF_Sample03.Mappings
{
    public class InterestComponentConfig : ComplexTypeConfiguration<InterestComponent>
    {
        public InterestComponentConfig()
        {
            this.Property(x => x.Interest1).HasMaxLength(450);
            this.Property(x => x.Interest2).HasMaxLength(450);
        }
    }
}
```

```
using System.Data.Entity.ModelConfiguration;
using EF_Sample03.Models;

namespace EF_Sample03.Mappings
{
    public class ProjectConfig : EntityTypeConfiguration<Project>
    {
        public ProjectConfig()
        {
            this.Property(x => x.Description).IsMaxLength();
            this.Property(x => x.RowVesrion).IsRowVersion();
        }
    }
}
```

```
using System.Data.Entity.ModelConfiguration;
using EF_Sample03.Models;
using System.ComponentModel.DataAnnotations;

namespace EF_Sample03.Mappings
{
```

```
    public class UserConfig : EntityTypeConfiguration<User>
    {
        public UserConfig()
        {
            this.HasKey(x => x.Id);
            this.Property(x => x.Id).HasDatabaseGeneratedOption(DatabaseGeneratedOption.Identity);
            this.ToTable("tblUser", schemaName: "guest");
            this.Property(p =>
p.AddDate).HasColumnName("CreateDate").HasColumnType("date").IsRequired();
            this.Property(x => x.Name).HasMaxLength(450);
            this.Property(x => x.LastName).IsMaxLength().IsConcurrencyToken();
            this.Property(x => x.Email).IsFixedLength().HasMaxLength(255); //nchar(128)
            this.Property(x => x.Photo).IsOptional();
            this.Property(x => x.RowVersion).IsRowVersion();
            this.Ignore(x => x.FullName);
        }
    }
}
```

توضیحاتی در مورد کلاس‌های تنظیمات نگاشت‌های خواص به جداول و فیلدهای بانک اطلاعاتی

نظم بخشیدن به تعاریف نگاشت‌ها

همانطور که ملاحظه می‌کنید، جهت نظم بیشتر پروژه و شلوغ نشدن متد OnModelCreating کلاس Context برنامه، که در ادامه کدهای آن معرفی خواهد شد، به ازای هر کلاس مدل، یک کلاس تنظیمات نگاشت‌ها را اضافه کرده‌ایم.
کلاس‌های معمولی نگاشت‌ها ازکلاس EntityTypeConfiguration مشتق خواهند شد و جهت تعریف کلاس InterestComponent به عنوان Complex Type، اینبار از کلاس ComplexTypeConfiguration ارث بری شده است.

تعیین طول فیلدها

در کلاس InterestComponentConfig، به کمک متد HasMaxLength، همان کار ویژگی MaxLength را می‌توان شبیه سازی کرد که در نهایت، طول فیلد nvarchar تشکیل شده در بانک اطلاعاتی را مشخص می‌کند. اگر نیاز است این فیلد nvarchar از نوع max باشد، نیازی به تنظیم خاصی نداشته و حالت پیش فرض است یا اینکه می‌توان صریحا از متد IsMaxLength نیز برای معرفی nvarchar max استفاده کرد.

تعیین مسایل همزمانی

در قسمت سوم با ویژگی‌های ConcurrencyCheck و Timestamp آشنا شدیم. در اینجا اگر نوع خاصیت byte array بود و نیاز به تعریف آن به صورت timestamp وجود داشت، می‌توان از متد IsRowVersion استفاده کرد. معادل ویژگی ConcurrencyCheck در اینجا، متد IsConcurrencyToken است.

تعیین کلید اصلی جدول

اگر پیش فرض‌های EF Code first مانند وجود خاصیتی به نام Id یا ClassName+Id رعایت شود، نیازی به کار خاصی نخواهد بود. اما اگر این قراردادها رعایت نشوند، می‌توان از متد HasKey (که نمونه‌ای از آن‌را در کلاس UserConfig فوق مشاهده می‌کنید)، استفاده کرد.

تعیین فیلدهای تولید شده توسط بانک اطلاعاتی

به کمک متد HasDatabaseGeneratedOption، می‌توان مشخص کرد که آیا یک فیلد Identity است و یا یک فیلد محاسباتی ویژه و یا هیچکدام.

تعیین نام جدول و schema آن

اگر نیاز است از قراردادهای نامگذاری خاصی پیروی شود، می‌توان از متد ToTable جهت تعریف نام جدول متناظر با کلاس جاری استفاده کرد. همچنین در اینجا امکان تعریف schema نیز وجود دارد.

تعیین نام و نوع سفارشی فیلدها

همچنین اگر نام فیلدها نیز باید از قراردادهای دیگری پیروی کنند، می‌توان آن‌ها را به صورت صریح توسط متد HasColumnName معرفی کرد. اگر نیاز است این خاصیت به نوع خاصی در بانک اطلاعاتی نگاشت شود، باید از متد HasColumnType کمک گرفت. برای مثال در اینجا بجای نوع datetime، از نوع ویژه date استفاده شده است.

معرفی فیلدها به صورت nchar بجای nvarchar

برای نمونه اگر قرار است هش کلمه عبور در بانک اطلاعاتی ذخیره شود، چون طول آن ثابت می‌باشد، توصیه شده‌است که بجای nvarchar از nchar برای تعریف آن استفاده شود. برای این منظور تنها کافی است از متد IsFixedLength استفاده شود. در این حالت طول پیش فرض 128 برای فیلد درنظر گرفته خواهد شد. بنابراین اگر نیاز است از طول دیگری استفاده شود، می‌توان همانند سابق از متد HasMaxLength کمک گرفت.

ضمنا این فیلدها همگی یونیکد هستند و با n شروع شده‌اند. اگر می‌خواهید از varchar یا char استفاده کنید، می‌توان از متد IsUnicode با پارامتر false استفاده کرد.

معرفی یک فیلد به صورت null پذیر در سمت بانک اطلاعاتی

استفاده از متد IsOptional، فیلد را در سمت بانک اطلاعاتی به صورت فیلدی با امکان پذیرش مقادیر null معرفی می‌کند.

البته در اینجا به صورت پیش فرض byte arrayها به همین نحو معرفی می‌شوند و تنظیم فوق صرفا جهت ارائه توضیحات بیشتر در نظر گرفته شد.

صرفنظر کردن از خواص محاسباتی در تعاریف نگاشت‌ها

با توجه به اینکه خاصیت FullName به صورت یک خاصیت محاسباتی فقط خواندنی، در کدهای برنامه تعریف شده است، با استفاده از متد Ignore، از نگاشت آن به بانک اطلاعاتی جلوگیری خواهیم کرد.

معرفی کلاس‌های تعاریف نگاشت‌ها به برنامه

استفاده از کلاس‌های Config فوق خودکار نیست و نیاز است توسط متد modelBuilder.Configurations.Add معرفی شوند:

```
using System.Data.Entity;
using System.Data.Entity.Migrations;
using EF_Sample03.Mappings;
using EF_Sample03.Models;

namespace EF_Sample03.DataLayer
{
    public class Sample03Context : DbContext
    {
        public DbSet<User> Users { set; get; }
        public DbSet<Project> Projects { set; get; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Configurations.Add(new InterestComponentConfig());
            modelBuilder.Configurations.Add(new ProjectConfig());
            modelBuilder.Configurations.Add(new UserConfig());

            //modelBuilder.ComplexType<InterestComponent>();
            //modelBuilder.Ignore<InterestComponent>();

            base.OnModelCreating(modelBuilder);
        }
    }

    public class Configuration : DbMigrationsConfiguration<Sample03Context>
    {
        public Configuration()
        {
            AutomaticMigrationsEnabled = true;
            AutomaticMigrationDataLossAllowed = true;
        }

        protected override void Seed(Sample03Context context)
        {
            base.Seed(context);
```

```
        }
    }
}
```

در اینجا کلاس Context برنامه مثال جاری را ملاحظه می‌کنید؛ به همراه کلاس Configuration مهاجرت خودکار که در قسمت‌های قبل بررسی شد.
در متد OnModelCreating نیز می‌توان یک کلاس را از نوع Complex معرفی کرد تا برای آن در بانک اطلاعاتی جدول جداگانه‌ای تعریف نشود. اما باید دقت داشت که اینکار را فقط یکبار می‌توان انجام داد؛ یا توسط کلاس InterestComponentConfig و یا توسط متد modelBuilder.ComplexType. اگر هر دو با هم فراخوانی شوند، EF یک استثناء را صادر خواهد کرد.

و در نهایت، قسمت آغازین برنامه اینبار به شکل زیر خواهد بود که از آغاز کننده MigrateDatabaseToLatestVersion (قسمت چهارم این سری) نیز استفاده کرده است:

```
using System;
using System.Data.Entity;
using EF_Sample03.DataLayer;

namespace EF_Sample03
{
    class Program
    {
        static void Main(string[] args)
        {
            Database.SetInitializer(new MigrateDatabaseToLatestVersion<Sample03Context,
Configuration>());

            using (var db = new Sample03Context())
            {
                var project1 = db.Projects.Find(1);
                if (project1 != null)
                {
                    Console.WriteLine(project1.Title);
                }
            }
        }
    }
}
```

ضمنا رشته اتصالی مورد استفاده تعریف شده در فایل کانفیگ برنامه نیز به صورت زیر تعریف شده است:

```
<connectionStrings>
    <clear/>
    <add
       name="Sample03Context"
       connectionString="Data Source=(local);Initial Catalog=testdb2012;Integrated Security = true"
       providerName="System.Data.SqlClient"
      />
/connectionStrings>
```

در قسمت‌های بعد مباحث پیشرفته‌تری از تنظیمات نگاشت‌ها را به کمک Fluent API، بررسی خواهیم کرد. برای مثال روابط ارث بری، many-to-many و ... چگونه تعریف می‌شوند.

عنوان: EF Code First #7
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۲۰ ۱۱:۰۱:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Entity framework

مدیریت روابط بین جداول در EF Code first به کمک Fluent API

EF Code first بجای اتلاف وقت شما با نوشتن فایل‌های XML تهیه نگاشت‌ها یا تنظیم آن‌ها با کد، رویه Convention over configuration را پیشنهاد می‌دهد. همین رویه، جهت مدیریت روابط بین جداول نیز برقرار است. روابط one-to-one، one-to-many، many-to-many و موارد دیگر را بدون یک سطر تنظیم اضافی، صرفا بر اساس یک سری قراردادهای توکار می‌تواند تشخیص داده و اعمال کند. عموما زمانی نیاز به تنظیمات دستی وجود خواهد داشت که قراردادهای توکار رعایت نشوند و یا برای مثال قرار است با یک بانک اطلاعاتی قدیمی از پیش موجود کار کنیم.

مفاهیمی به نام‌های Principal و Dependent

در EF Code first از یک سری واژه‌های خاص جهت بیان ابتدا و انتهای روابط استفاده شده است که عدم آشنایی با آن‌ها درک خطاهای حاصل را مشکل می‌کند:
الف) Principal : طرفی از رابطه است که ابتدا در بانک اطلاعاتی ذخیره خواهد شد.
ب) Dependent : طرفی از رابطه است که پس از ثبت Principal در بانک اطلاعاتی ذخیره می‌شود.
Principal می‌تواند بدون نیاز به Dependent وجود داشته باشد. وجود Dependent بدون Principal ممکن نیست زیرا ارتباط بین این دو توسط یک کلید خارجی تعریف می‌شود.

کدهای مثال مدیریت روابط بین جداول

در دنیای واقعی، همه‌ی مثال‌ها به مدل بلاگ و مطالب آن ختم نمی‌شوند. به همین جهت نیاز است یک مدل نسبتا پیچیده‌تر را در اینجا بررسی کنیم. در ادامه کدهای کامل مثال جاری را مشاهده خواهید کرد:

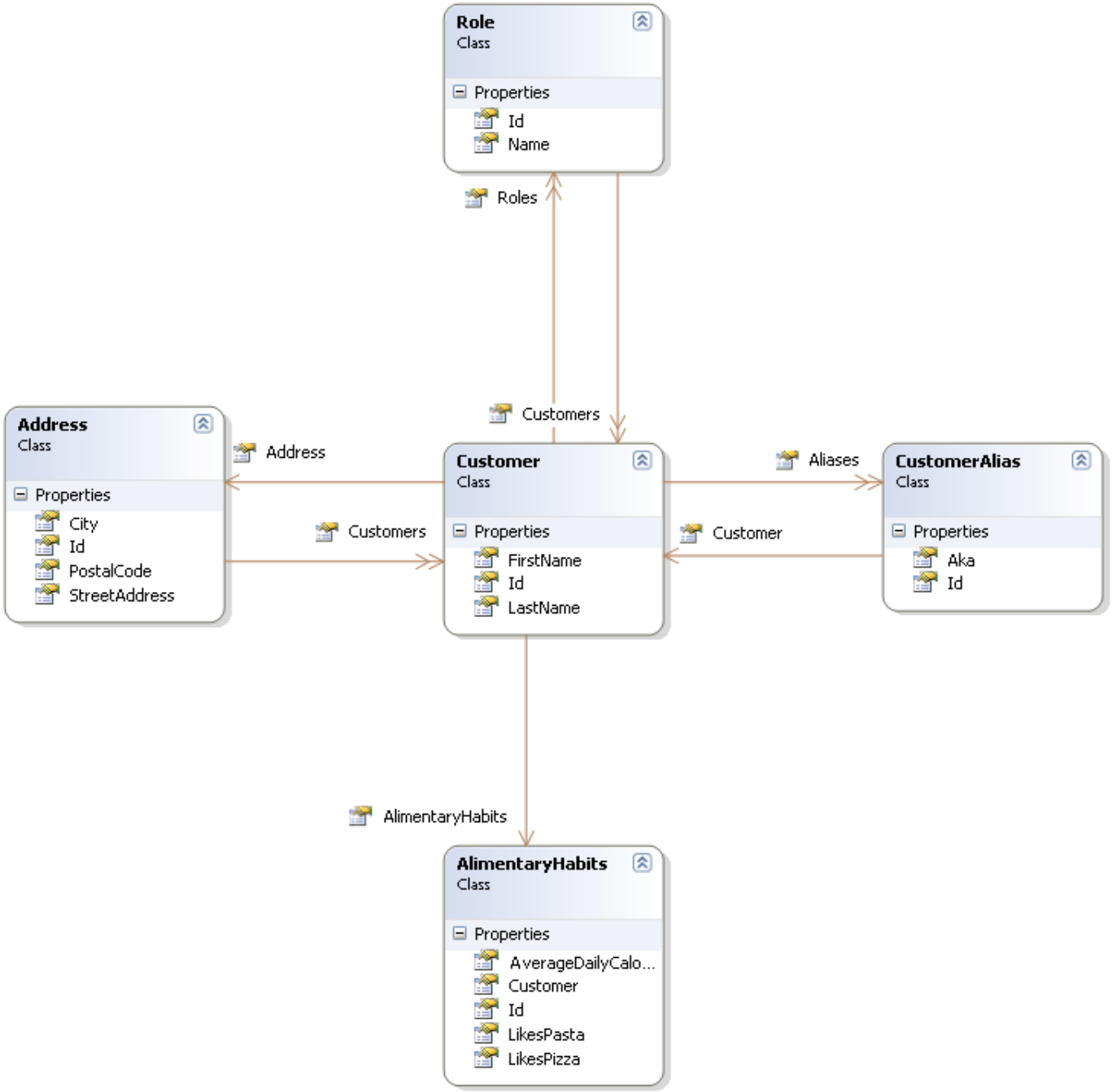
```
using System.Collections.Generic;

namespace EF_Sample35.Models
{
    public class Customer
    {
        public int Id { set; get; }
        public string FirstName { set; get; }
        public string LastName { set; get; }

        public virtual AlimentaryHabits AlimentaryHabits { set; get; }
        public virtual ICollection<CustomerAlias> Aliases { get; set; }
        public virtual ICollection<Role> Roles { get; set; }
        public virtual Address Address { get; set; }
    }
}
```

```
namespace EF_Sample35.Models
{
    public class CustomerAlias
    {
        public int Id { get; set; }
        public string Aka { get; set; }

        public virtual Customer Customer { get; set; }
    }
}
```

```
using System.Collections.Generic;

namespace EF_Sample35.Models
{
    public class Role
    {
        public int Id { set; get; }
        public string Name { set; get; }

        public virtual ICollection<Customer> Customers { set; get; }
    }
}
```

```
namespace EF_Sample35.Models
{
    public class AlimentaryHabits
    {
        public int Id { get; set; }
        public bool LikesPasta { get; set; }
        public bool LikesPizza { get; set; }
        public int AverageDailyCalories { get; set; }

        public virtual Customer Customer { get; set; }
    }
}
```

```
using System.Collections.Generic;

namespace EF_Sample35.Models
{
    public class Address
    {
        public int Id { set; get; }
        public  string City { set; get; }
        public  string StreetAddress { set; get; }
        public  string PostalCode { set; get; }

        public virtual ICollection<Customer> Customers { set; get; }
    }
}
```



شکل ۱

همچنین **تعاریف نگاشت‌های برنامه** نیز مطابق کدهای زیر است:

```
using System.Data.Entity.ModelConfiguration;
using EF_Sample35.Models;

namespace EF_Sample35.Mappings
{
    public class CustomerAliasConfig : EntityTypeConfiguration<CustomerAlias>
    {
        public CustomerAliasConfig()
        {
            // one-to-many
            this.HasRequired(x => x.Customer)
                .WithMany(x => x.Aliases)
```

```
                .WillCascadeOnDelete();
        }
    }
}
```

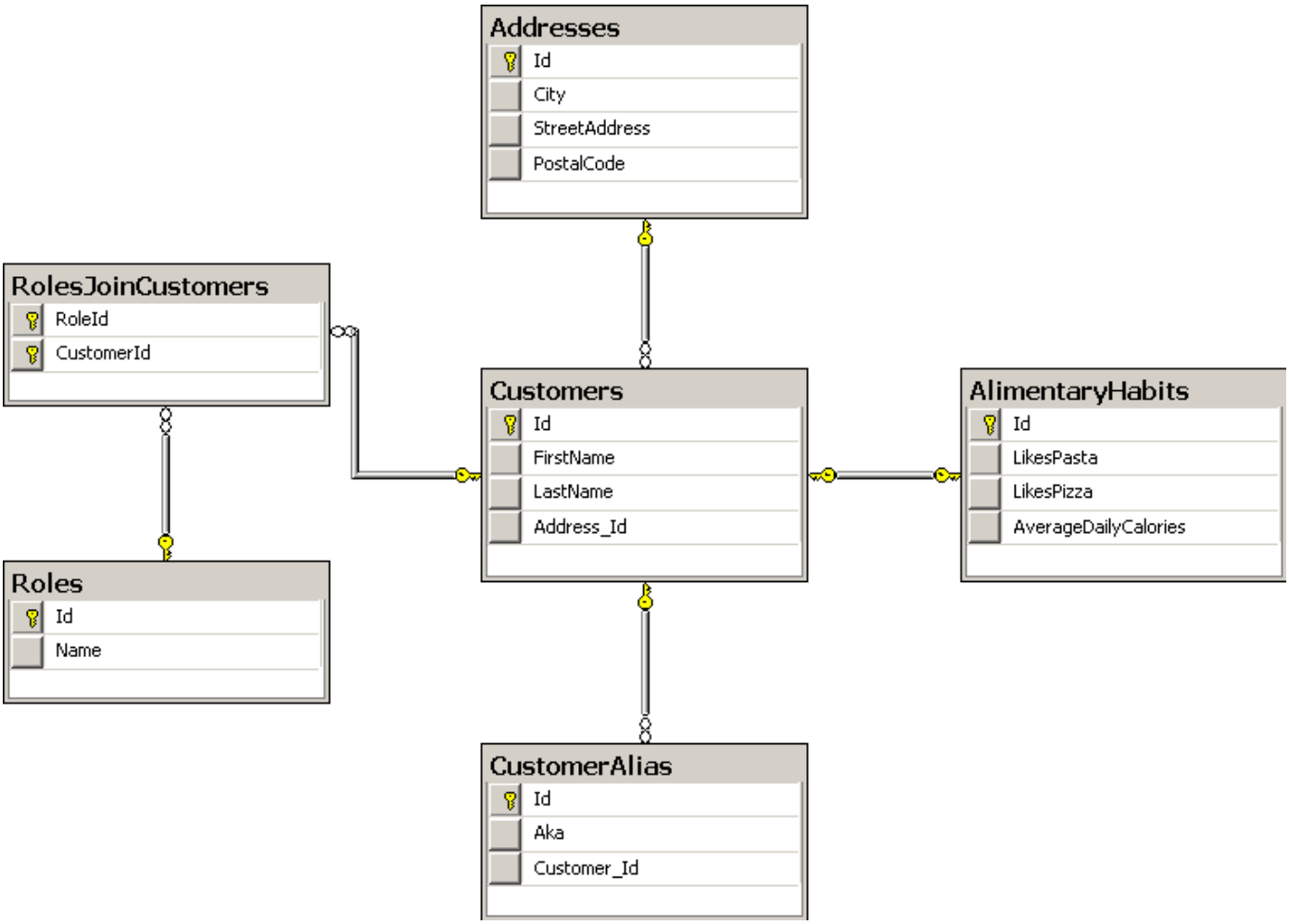
```
using System.Data.Entity.ModelConfiguration;
using EF_Sample35.Models;

namespace EF_Sample35.Mappings
{
    public class CustomerConfig : EntityTypeConfiguration<Customer>
    {
        public CustomerConfig()
        {
            // one-to-one
            this.HasOptional(x => x.AlimentaryHabits)
                .WithRequired(x => x.Customer)
                .WillCascadeOnDelete();

            // many-to-many
            this.HasMany(p => p.Roles)
                .WithMany(t => t.Customers)
                .Map(mc =>
                   {
                       mc.ToTable("RolesJoinCustomers");
                       mc.MapLeftKey("RoleId");
                       mc.MapRightKey("CustomerId");
                   });

            // many-to-one
            this.HasOptional(x => x.Address)
                .WithMany(x => x.Customers)
                .WillCascadeOnDelete();
        }
    }
}
```

به همراه **Context** زیر:

```
using System.Data.Entity;
using System.Data.Entity.Migrations;
using EF_Sample35.Mappings;
using EF_Sample35.Models;

namespace EF_Sample35.DataLayer
{
    public class Sample35Context : DbContext
    {
        public DbSet<AlimentaryHabits> AlimentaryHabits { set; get; }
        public DbSet<Customer> Customers { set; get; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Configurations.Add(new CustomerConfig());
            modelBuilder.Configurations.Add(new CustomerAliasConfig());

            base.OnModelCreating(modelBuilder);
        }
    }

    public class Configuration : DbMigrationsConfiguration<Sample35Context>
    {
        public Configuration()
        {
            AutomaticMigrationsEnabled = true;
            AutomaticMigrationDataLossAllowed = true;
        }

        protected override void Seed(Sample35Context context)
        {
```

```
            base.Seed(context);
        }
    }
}
```

که نهایتا منجر به تولید چنین **ساختاری در بانک اطلاعاتی** می‌گردد:



شکل ۲

توضیحات کامل کدهای فوق:

تنظیمات روابط one-to-one و یا one-to-zero

زمانیکه رابطه‌ای 1..0 و یا 1..1 است، مطابق قراردادهای توکار EF Code first تنها کافی است یک navigation property را که بیانگر ارجاعی است به شیء دیگر، تعریف کنیم (در هر دو طرف رابطه).

برای مثال در مدل‌های فوق یک مشتری که در حین ثبت اطلاعات اصلی او، «ممکن است» اطلاعات جانبی دیگری (AlimentaryHabits) نیز از او تنها در طی یک رکورد، دریافت شود. قصد هم نداریم یک ComplexType را تعریف کنیم. نیاز است جدول AlimentaryHabits جداگانه وجود داشته باشد.

```
namespace EF_Sample35.Models
{
    public class Customer
    {
        // ...
        public virtual AlimentaryHabits AlimentaryHabits { set; get; }
    }
}
```

```
namespace EF_Sample35.Models
{
    public class AlimentaryHabits
    {
        // ...
        public virtual Customer Customer { get; set; }
    }
}
```

در اینجا خواص virtual تعریف شده در دو طرف رابطه، به EF خواهد گفت که رابطه‌ای، 1:1 برقرار است. در این حالت اگر برنامه را اجرا کنیم، به خطای زیر برخواهیم خورد:

```
Unable to determine the principal end of an association between
the types 'EF_Sample35.Models.Customer' and 'EF_Sample35.Models.AlimentaryHabits'.
The principal end of this association must be explicitly configured using either
the relationship fluent API or data annotations.
```

EF تشخیص داده است که رابطه 1:1 برقرار است؛ اما با قاطعیت نمی‌تواند طرف Principal را تعیین کند. بنابراین باید اندکی به او کمک کرد:

```
using System.Data.Entity.ModelConfiguration;
using EF_Sample35.Models;

namespace EF_Sample35.Mappings
{
    public class CustomerConfig : EntityTypeConfiguration<Customer>
    {
        public CustomerConfig()
        {
            // one-to-one
            this.HasOptional(x => x.AlimentaryHabits)
                .WithRequired(x => x.Customer)
                .WillCascadeOnDelete();
        }
    }
}
```

همانطور که ملاحظه می‌کنید در اینجا توسط متد WithRequired طرف Principal و توسط متد HasOptional، طرف Dependent تعیین شده است. به این ترتیب EF می‌توان یک رابطه 1:1 را تشکیل دهید.
توسط متد WillCascadeOnDelete هم مشخص می‌کنیم که اگر Principal حذف شد، لطفا Dependent را به صورت خودکار حذف کن.

توضیحات ساختار جداول تشکیل شده:

هر دو جدول با همان خواص اصلی که در دو کلاس وجود دارند، تشکیل شده‌اند.
فیلد Id جدول AlimentaryHabits اینبار دیگر Identity نیست. اگر به تعریف قید FK_AlimentaryHabits_Customers_Id دقت کنیم، در اینجا مشخص است که فیلد Id جدول AlimentaryHabits، به فیلد Id جدول مشتری‌ها متصل شده است (یعنی در آن واحد هم primary key است و هم foreign key). به همین جهت به این روش one-to-one association with shared primary key هم گفته می‌شود (کلید اصلی جدول مشتری با جدول AlimentaryHabits به اشتراک گذاشته شده است).

تنظیمات روابط one-to-many

برای مثال همان مشتری فوق را درنظر بگیرید که دارای تعدادی نام مستعار است:

```
using System.Collections.Generic;

namespace EF_Sample35.Models
{
    public class Customer
    {
        // ...
        public virtual ICollection<CustomerAlias> Aliases { get; set; }
    }
}
```

```
namespace EF_Sample35.Models
{
    public class CustomerAlias
    {
        // ...
        public virtual Customer Customer { get; set; }
    }
}
```

همین میزان تنظیم کفایت می‌کند و نیازی به استفاده از Fluent API برای معرفی روابط نیست.
در طرف Principal، یک مجموعه یا لیستی از Dependent وجود دارد. در Dependent هم یک navigation property معرف طرف Principal اضافه شده است.
جدول CustomerAlias اضافه شده، توسط یک کلید خارجی به جدول مشتری مرتبط می‌شود.

سؤال: اگر در اینجا نیز بخواهیم CascadeOnDelete را اعمال کنیم، چه باید کرد؟
پاسخ: جهت سفارشی سازی نحوه تعاریف روابط حتما نیاز به استفاده از Fluent API به نحو زیر می‌باشد:

```
using System.Data.Entity.ModelConfiguration;
using EF_Sample35.Models;

namespace EF_Sample35.Mappings
{
    public class CustomerAliasConfig : EntityTypeConfiguration<CustomerAlias>
    {
        public CustomerAliasConfig()
        {
            // one-to-many
            this.HasRequired(x => x.Customer)
                .WithMany(x => x.Aliases)
                .WillCascadeOnDelete();
        }
    }
}
```

اینکار را باید در کلاس تنظیمات CustomerAlias انجام داد تا بتوان Principal را توسط متد HasRequired به Customer و سپس dependent را به کمک متد WithMany مشخص کرد. در ادامه می‌توان متد WillCascadeOnDelete یا هر تنظیم سفارشی دیگری را نیز اعمال نمود.
متد HasRequired سبب خواهد شد فیلد Customer_Id، به صورت not null در سمت بانک اطلاعاتی تعریف شود؛ متد HasOptional عکس آن است.

تنظیمات روابط many-to-many

برای تنظیم روابط many-to-many تنها کافی است دو سر رابطه ارجاعاتی را به یکدیگر توسط یک لیست یا مجموعه داشته باشند:

```
using System.Collections.Generic;

namespace EF_Sample35.Models
{
    public class Role
    {
        // ...
        public virtual ICollection<Customer> Customers { set; get; }
    }
}
```

```
using System.Collections.Generic;

namespace EF_Sample35.Models
{
    public class Customer
    {
        // ...
        public virtual ICollection<Role> Roles { get; set; }
    }
}
```

همانطور که مشاهده می‌کنید، یک مشتری می‌تواند چندین نقش داشته باشد و هر نقش می‌تواند به چندین مشتری منتسب شود.
اگر برنامه را به این ترتیب اجرا کنیم، به صورت خودکار یک رابطه many-to-many تشکیل خواهد شد (بدون نیاز به تنظیمات نگاشت‌های آن). نکته جالب آن تشکیل خودکار جدول ارتباط دهنده واسط یا اصطلاحا join-table می‌باشد:

```
CREATE TABLE [dbo].[RolesJoinCustomers](
 [RoleId] [int] NOT NULL,
 [CustomerId] [int] NOT NULL,
)
```

سؤال: نام‌های خودکار استفاده شده را می‌خواهیم تغییر دهیم. چکار باید کرد؟
پاسخ: اگر بانک اطلاعاتی برای بار اول است که توسط این روش تولید می‌شود شاید این پیش فرض‌ها اهمیتی نداشته باشد و نسبتا هم مناسب هستند. اما اگر قرار باشد از یک بانک اطلاعاتی موجود که امکان تغییر نام فیلدها و جداول آن وجود ندارد استفاده کنیم، نیاز به سفارشی سازی تعاریف نگاشت‌ها به کمک Fluent API خواهیم داشت:

```
using System.Data.Entity.ModelConfiguration;
using EF_Sample35.Models;

namespace EF_Sample35.Mappings
{
    public class CustomerConfig : EntityTypeConfiguration<Customer>
    {
```

```
        public CustomerConfig()
        {
            // many-to-many
            this.HasMany(p => p.Roles)
                .WithMany(t => t.Customers)
                .Map(mc =>
                   {
                       mc.ToTable("RolesJoinCustomers");
                       mc.MapLeftKey("RoleId");
                       mc.MapRightKey("CustomerId");
                   });
        }
    }
}
```

تنظیمات روابط many-to-one

در تکمیل مدل‌های مثال جاری، به دو کلاس زیر خواهیم رسید. در اینجا تنها در کلاس مشتری است که ارجاعی به کلاس آدرس او وجود دارد. در کلاس آدرس، یک navigation property همانند حالت 1:1 تعریف نشده است:

```
namespace EF_Sample35.Models
{
    public class Address
    {
        public int Id { set; get; }
        public  string City { set; get; }
        public  string StreetAddress { set; get; }
        public  string PostalCode { set; get; }
    }
}
```

```
using System.Collections.Generic;

namespace EF_Sample35.Models
{
    public class Customer
    {
       // …
        public virtual Address Address { get; set; }
    }
}
```

این رابطه توسط EF Code first به صورت خودکار به یک رابطه many-to-one تفسیر خواهد شد و نیازی به تنظیمات خاصی ندارد. زمانیکه جداول برنامه تشکیل شوند، جدول Addresses موجودیتی مستقل خواهد داشت و جدول مشتری با یک فیلد به نام Address_Id به جدول آدرس‌ها متصل می‌گردد. این فیلد نال پذیر است؛ به عبارتی ذکر آدرس مشتری الزامی نیست.
اگر نیاز بود این تعاریف نیز توسط Fluent API سفارشی شوند، باید خاصیت public virtual ICollection<Customer> Customers به کلاس Address نیز اضافه شود تا بتوان رابطه زیر را توسط کدهای برنامه تعریف کرد:

```
using System.Data.Entity.ModelConfiguration;
using EF_Sample35.Models;

namespace EF_Sample35.Mappings
{
    public class CustomerConfig : EntityTypeConfiguration<Customer>
    {
        public CustomerConfig()
        {
            // many-to-one
```

```
            this.HasOptional(x => x.Address)
                .WithMany(x => x.Customers)
                .WillCascadeOnDelete();
        }
    }
}
```

متد HasOptional سبب می‌شود تا فیلد Address_Id اضافه شده به جدول مشتری‌ها، null پذیر شود.

عنوان: EF Code First #8
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۲۱ ۱۳:۴۷:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Entity framework

ادامه بحث بررسی جزئیات نحوه نگاشت کلاس‌ها به جداول، توسط EF Code first

استفاده از Viewهای SQL Server در EF Code first

از Viewها عموما همانند یک جدول فقط خواندنی استفاده می‌شود. بنابراین نحوه نگاشت اطلاعات یک کلاس به یک View دقیقا همانند نحوه نگاشت اطلاعات یک کلاس به یک جدول است و تمام نکاتی که تا کنون بررسی شدند، در اینجا نیز صادق است. اما ...
الف) بر اساس تنظیمات توکار EF Code first، نام مفرد کلاس‌ها، حین نگاشت به جداول، تبدیل به اسم جمع می‌شوند. بنابراین اگر View ما در سمت بانک اطلاعاتی چنین تعریفی دارد:

```
Create VIEW EmployeesView
AS
SELECT id,
       FirstName
FROM   Employees
```

در سمت کدهای برنامه نیاز است به این شکل تعریف شود:

```
using System.ComponentModel.DataAnnotations;

namespace EF_Sample04.Models
{
    [Table("EmployeesView")]
    public class EmployeesView
    {
        public int Id { set; get; }
        public string FirstName { set; get; }
    }
}
```

در اینجا به کمک ویژگی Table، نام دقیق این View را در بانک اطلاعاتی مشخص کرده‌ایم. به این ترتیب تنظیمات توکار EF بازنویسی خواهد شد و دیگر به دنبال EmployeesViews نخواهد گشت؛ یا جدول متناظر با آن‌را به صورت خودکار ایجاد نخواهد کرد.
ب) View شما نیاز است دارای یک فیلد Primary key نیز باشد.
ج) اگر از مهاجرت خودکار توسط MigrateDatabaseToLatestVersion استفاده کنیم، پیغام خطای زیر را دریافت خواهیم کرد:

```
There is already an object named 'EmployeesView' in the database.
```

علت این است که هنوز جدول dbo.__MigrationHistory از وجود آن مطلع نشده است، زیرا یک View، خارج از برنامه و در سمت بانک اطلاعاتی اضافه می‌شود.
برای حل این مشکل می‌توان همانطور که در قسمت‌های قبل نیز عنوان شد، EF را طوری تنظیم کرد تا کاری با بانک اطلاعاتی نداشته باشد:

```
Database.SetInitializer<Sample04Context>(null);
```

به این ترتیب EmployeesView در همین لحظه قابل استفاده است.
و یا به حالت امن مهاجرت دستی سوئیچ کنید:

```
Add-Migration Init -IgnoreChanges
Update-Database
```

پارامتر IgnoreChanges سبب می‌شود تا متدهای Up و Down کلاس مهاجرت تولید شده، خالی باشد. یعنی زمانیکه دستور Update-Database انجام می‌شود، نه Viewایی دراپ خواهد شد و نه جدول اضافه‌ای ایجاد می‌گردد. فقط جدول dbo.__MigrationHistory به روز می‌شود که هدف اصلی ما نیز همین است.
همچنین در این حالت کنترل کاملی بر روی کلاس‌های Up و Down وجود دارد. می‌توان CreateTable اضافی را به سادگی از این کلاس‌ها حذف کرد.

ضمن اینکه باید دقت داشت یکی از اهداف کار با ORMs، فراهم شدن امکان استفاده از بانک‌های اطلاعاتی مختلف، بدون اعمال تغییری در کدهای برنامه می‌باشد (فقط تغییر کانکشن استرینگ، به علاوه تعیین Provider جدید، باید جهت این مهاجرت کفایت کند). بنابراین اگر از View استفاده می‌کنید، این برنامه به SQL Server گره خواهد خورد و دیگر از سایر بانک‌های اطلاعاتی که از این مفهوم پشتیبانی نمی‌کنند، نمی‌توان به سادگی استفاده کرد.

استفاده از فیلدهای XML اس کیوال سرور

در حال حاضر پشتیبانی توکاری توسط EF Code first از فیلدهای ویژه XML اس کیوال سرور وجود ندارد؛ اما استفاده از آن‌ها با رعایت چند نکته ساده، به نحو زیر است:

```
using System.ComponentModel.DataAnnotations;
using System.Xml.Linq;

namespace EF_Sample04.Models
{
    public class MyXMLTable
    {
        public int Id { get; set; }

        [Column(TypeName = "xml")]
        public string XmlValue { get; set; }

        [NotMapped]
        public XElement XmlValueWrapper
        {
            get { return XElement.Parse(XmlValue); }
            set { XmlValue = value.ToString(); }
        }
    }
}
```

در اینجا توسط TypeName ویژگی Column، نوع توکار xml مشخص شده است. این فیلد در طرف کدهای کلاس‌های برنامه، به صورت string تعریف می‌شود. سپس اگر نیاز بود به این خاصیت توسط LINQ to XML دسترسی یافت، می‌توان یک فیلد محاسباتی را همانند خاصیت XmlValueWrapper فوق تعریف کرد. نکته دیگری را که باید به آن دقت داشت، استفاده از ویژگی NotMapped

می‌باشد، تا EF سعی نکند خاصیتی از نوع XElement را (یک CLR Property) به بانک اطلاعاتی نگاشت کند.

و همچنین اگر علاقمند هستید که این قابلیت به صورت توکار اضافه شود، می‌توانید اینجا رای دهید !

نحوه تعریف Composite keys در EF Code first

کلاس نوع فعالیت زیر را درنظر بگیرید:

```
namespace EF_Sample04.Models
{
    public class ActivityType
    {
        public int UserId { set; get; }
        public int ActivityID { get; set; }
    }
}
```

در جدول متناظر با این کلاس، نباید دو رکورد تکراری حاوی شماره کاربری و شماره فعالیت یکسانی باهم وجود داشته باشند. بنابراین بهتر است بر روی این دو فیلد، یک کلید ترکیبی تعریف کرد:

```
using System.Data.Entity.ModelConfiguration;
using EF_Sample04.Models;

namespace EF_Sample04.Mappings
{
    public class ActivityTypeConfig : EntityTypeConfiguration<ActivityType>
    {
        public ActivityTypeConfig()
        {
            this.HasKey(x => new { x.ActivityID, x.UserId });
        }
    }
}
```

در اینجا نحوه معرفی بیش از یک کلید را در متد HasKey ملاحظه می‌کنید.

یک نکته:
اینبار اگر سعی کنیم مثلا از متد db.ActivityTypes.Find با یک پارامتر استفاده کنیم، پیغام خطای «The number of primary key values passed must match number of primary key values defined on the entity» را دریافت خواهیم کرد. برای رفع آن باید هر دو کلید، در این متد قید شوند:

```
var activity1 = db.ActivityTypes.Find(4, 1);
```

ترتیب آن‌ها هم بر اساس ترتیبی که در کلاس ActivityTypeConfig، ذکر شده است، مشخص می‌گردد. بنابراین در این مثال، اولین پارامتر متد Find، به ActivityID اشاره می‌کند و دومین پارامتر به UserId.

بررسی نحوه تعریف نگاشت جداول خود ارجاع دهنده (Self Referencing Entity)

سناریوهای کاربردی بسیاری را جهت جداول خود ارجاع دهنده می‌توان متصور شد و عموما تمام آن‌ها برای مدل سازی اطلاعات

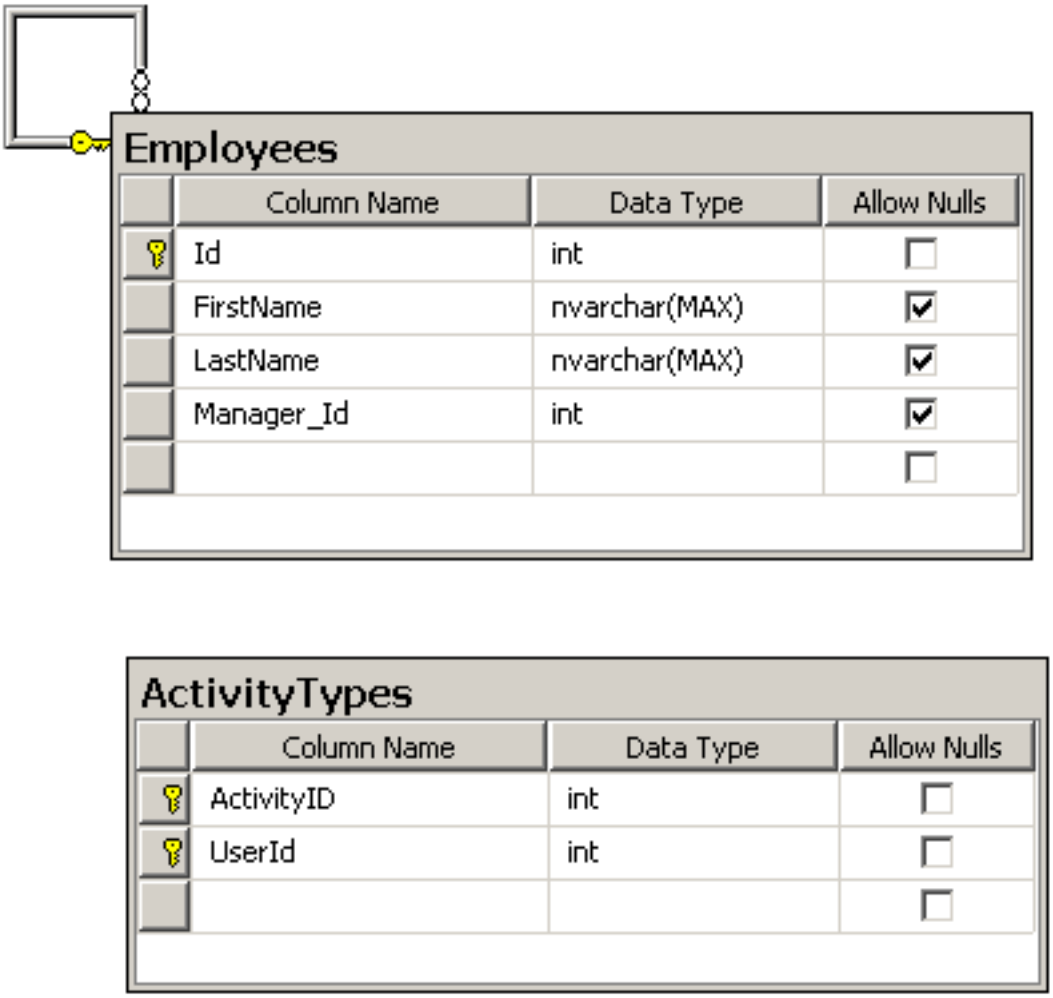
چند سطحی کاربرد دارند. برای مثال یک کارمند را درنظر بگیرید. مدیر این شخص هم یک کارمند است. مسئول این مدیر هم یک کارمند است و الی آخر. نمونه دیگر آن، طراحی منوهای چند سطحی هستند و یا یک مشتری را درنظر بگیرید. مشتری دیگری که توسط این مشتری معرفی شده است نیز یک مشتری است. این مشتری نیز می‌تواند یک مشتری دیگر را به شما معرفی کند و این سلسله مراتب به همین ترتیب می‌تواند ادامه پیدا کند.
در طراحی بانک‌های اطلاعاتی، برای ایجاد یک چنین جداولی، یک کلید خارجی را که به کلید اصلی همان جدول اشاره می‌کند، ایجاد خواهند کرد؛ اما در EF Code first چطور؟

```
using System.Collections.Generic;

namespace EF_Sample04.Models
{
    public class Employee
    {
        public int Id { set; get; }
        public string FirstName { get; set; }
        public string LastName { get; set; }

        //public int? ManagerID { get; set; }
        public virtual Employee Manager { get; set; }
    }
}
```

در این کلاس، خاصیت Manager دارای ارجاعی است به همان کلاس؛ یعنی یک کارمند می‌تواند مسئول کارمند دیگری باشد. برای تعریف نگاشت این کلاس به بانک اطلاعاتی می‌توان از روش زیر استفاده کرد:

```
using System.Data.Entity.ModelConfiguration;
using EF_Sample04.Models;

namespace EF_Sample04.Mappings
{
    public class EmployeeConfig : EntityTypeConfiguration<Employee>
    {
        public EmployeeConfig()
        {
            this.HasOptional(x => x.Manager)
                .WithMany()
                //.HasForeignKey(x => x.ManagerID)
                .WillCascadeOnDelete(false);
        }
    }
}
```

با توجه به اینکه یک کارمند می‌تواند مسئولی نداشته باشد (خودش مدیر ارشد است)، به کمک متد HasOptional مشخص کرده‌ایم که فیلد Manager_Id را که می‌خواهی به این کلاس اضافه کنی باید نال پذیر باشد. توسط متد WithMany طرف دیگر رابطه مشخص شده است.
اگر نیاز بود فیلد Manager_Id اضافه شده نام دیگری داشته باشد، یک خاصیت nullable مانند ManagerID را که در کلاس Employee مشاهده می‌کنید، اضافه نمائید. سپس در طرف تعاریف نگاشت‌ها به کمک متد HasForeignKey، باید صریحا عنوان کرد که این خاصیت، همان کلید خارجی است. از این نکته در سایر حالات تعاریف نگاشت‌ها نیز می‌توان استفاده کرد، خصوصا اگر از یک بانک اطلاعاتی موجود قرار است استفاده شود و از نام‌های دیگری بجز نام‌های پیش فرض EF استفاده کرده است.



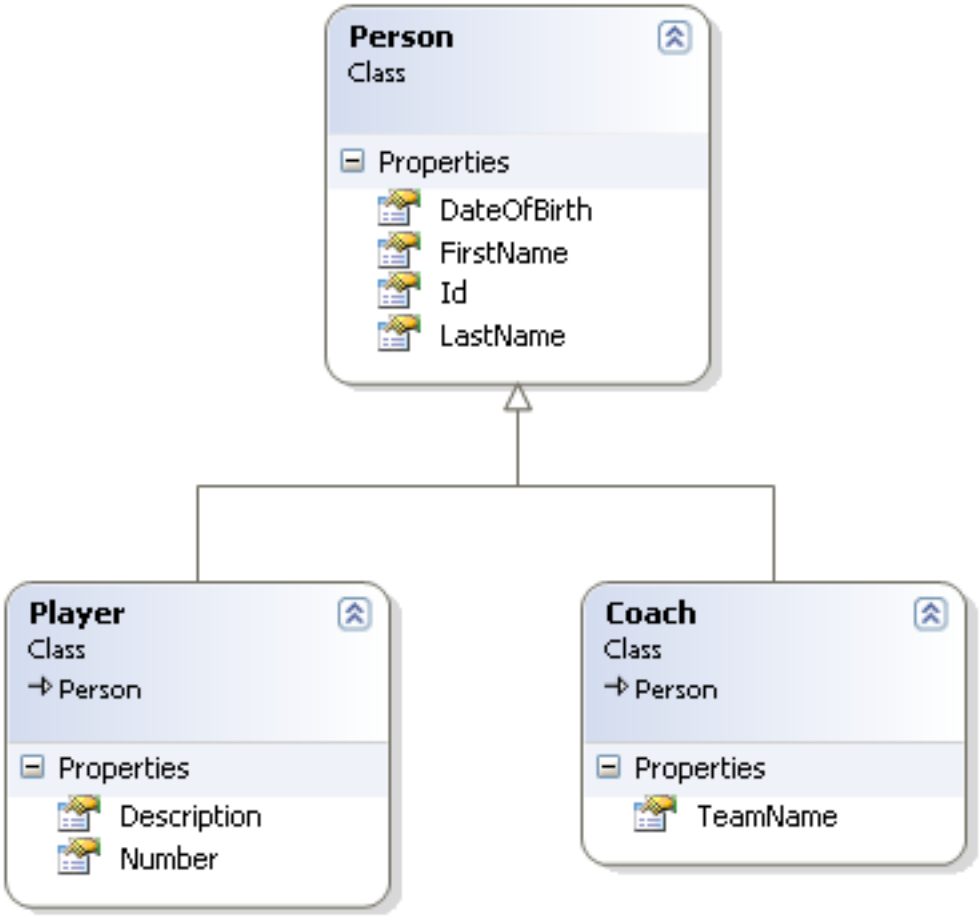
شکل ۱

مثال‌های این سری رو از این آدرس هم می‌تونید دریافت کنید: (^)

عنوان: EF Code First #9
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۲۲ ۱۳:۴۷:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Entity framework

تنظیمات ارث بری کلاس‌ها در EF Code first

بانک‌های اطلاعاتی مبتنی بر SQL، تنها روابطی از نوع «has a» یا «دارای» را پشتیبانی می‌کنند؛ اما در دنیای شیءگرا روابطی مانند «is a» یا «هست» نیز قابل تعریف هستند. برای توضیحات بیشتر به مدل‌های زیر دقت نمائید:



شکل ۱

```
using System;

namespace EF_Sample05.DomainClasses.Models
{
    public abstract class Person
    {
        public int Id { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime DateOfBirth { get; set; }
    }
}
```

```
namespace EF_Sample05.DomainClasses.Models
{
    public class Coach : Person
    {
        public string TeamName { set; get; }
    }
}
```

```
namespace EF_Sample05.DomainClasses.Models
{
    public class Player : Person
    {
        public int Number { get; set; }
        public string Description { get; set; }
    }
}
```

در این مدل‌ها که بر اساس ارث بری از کلاس شخص، تهیه شده‌اند؛ بازیکن، یک شخص است. مربی نیز یک شخص است؛ و به این ترتیب خوانده می‌شوند:

```
Coach "is a" Person
Player "is a" Person
```

در EF Code first سه روش جهت کار با این نوع کلاس‌ها و کلا ارث بری وجود دارد که در ادامه به آن‌ها خواهیم پرداخت:

الف) Table per Hierarchy یا TPH

همانطور که از نام آن نیز پیدا است، کل سلسله مراتبی را که توسط ارث بری تعریف شده است، تبدیل به یک جدول در بانک اطلاعاتی می‌کند. این حالت، شیوه برخورد پیش فرض EF Code first با ارث بری کلاس‌ها است و نیاز به هیچگونه تنظیم خاصی ندارد.
برای آزمایش این مساله، کلاس Context را به نحو زیر تعریف نمائید و سپس اجازه دهید تا EF بانک اطلاعاتی معادل آن‌را تولید کند:

```
using System.Data.Entity;
using EF_Sample05.DomainClasses.Models;

namespace EF_Sample05.DataLayer.Context
{
    public class Sample05Context : DbContext
    {
        public DbSet<Person> People { set; get; }
    }
}
```

ساختار جدول تولید شده آن همانند تصویر زیر است:

People

Column Name	Data Type	Allow Nulls
Id	int	☐
FirstName	nvarchar(MAX)	☑
LastName	nvarchar(MAX)	☑
DateOfBirth	datetime	☐
Number	int	☑
Description	nvarchar(MAX)	☑
TeamName	nvarchar(MAX)	☑
Discriminator	nvarchar(128)	☐
		☐

شکل ۲

همانطور که ملاحظه می‌کنید، تمام کلاس‌های مشتق شده از کلاس شخص را تبدیل به یک جدول کرده است؛ به علاوه یک فیلد جدید را هم به نام Discriminator به این جدول اضافه نموده است. برای درک بهتر عملکرد این فیلد، چند رکورد را توسط برنامه به بانک اطلاعاتی اضافه می‌کنیم. حاصل آن به شکل زیر خواهد بود:

Results | Messages

	Id	FirstName	LastName	DateOfBirth	Number	Description	TeamName	Discriminator
1	1	Coach F1	Coach L1	1962-05-11 10:59:25.853	NULL	NULL	Team A	Coach
2	2	Coach F2	Coach L2	1962-05-11 10:59:29.883	NULL	NULL	Team B	Coach
3	3	Coach F1	Coach L1	1962-05-11 10:59:29.883	1	...	NULL	Player

شکل ۳

از فیلد Discriminator جهت ثبت نام کلاس‌های متناظر با هر رکورد، استفاده شده است. به این ترتیب EF حین کار با اشیاء دقیقا می‌داند که چگونه باید خواص متناظر با کلاس‌های مختلف را مقدار دهی کند.
به علاوه اگر به ساختار جدول تهیه شده دقت کنید، مشخص است که در حالت TPH، نیاز است فیلدهای متناظر با کلاس‌های مشتق شده از کلاس پایه، همگی null پذیر باشند. برای نمونه فیلد Number که از نوع int تعریف شده، در سمت بانک اطلاعاتی نال پذیر تعریف شده است.
و برای کوئری نوشتن در این حالت می‌توان از متد الحاقی OfType جهت فیلتر کردن اطلاعات بر اساس کلاسی خاص، کمک گرفت:

```
db.People.OfType<Coach>().FirstOrDefault(x => x.LastName == "Coach L1")
```

سفارشی سازی نحوه نگاشت TPH

همانطور که عنوان شد، TPH نیاز به تنظیمات خاصی ندارد و حالت پیش فرض است؛ اما برای مثال می‌توان بر روی مقادیر و نوع ستون Discriminator تولیدی، کنترل داشت. برای این منظور باید از Fluent API به نحو زیر استفاده کرد:

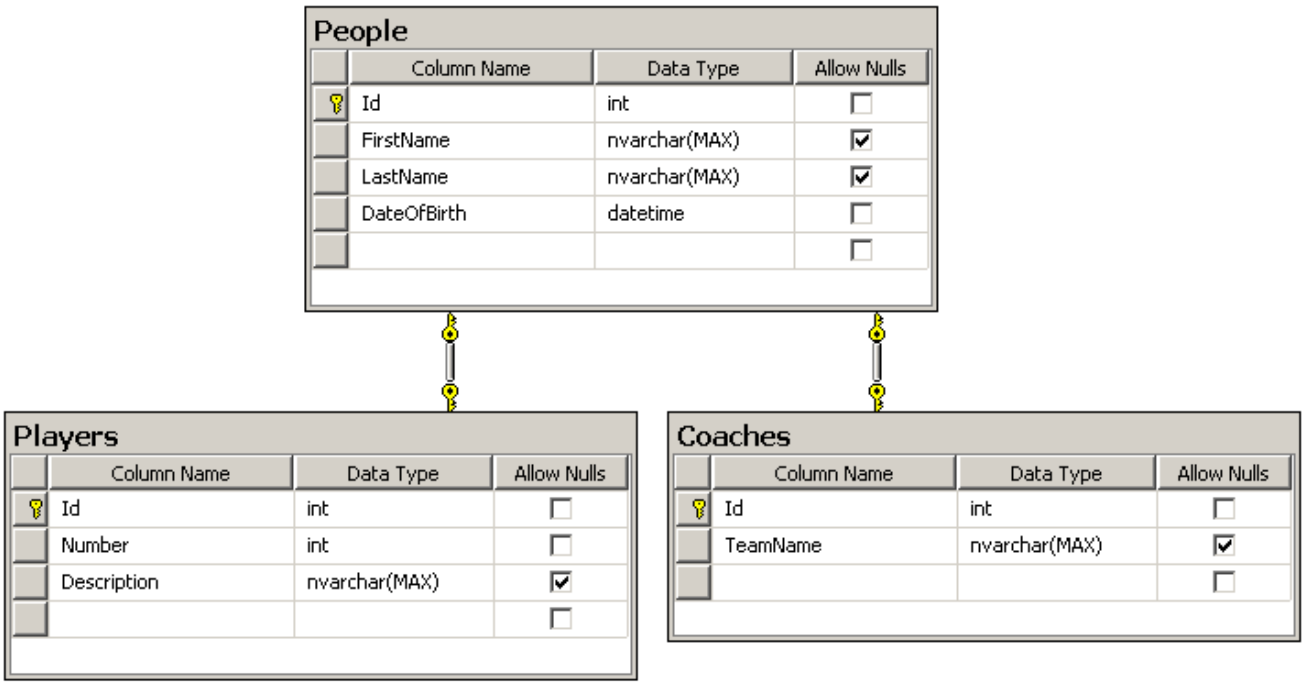
```
using System.Data.Entity.ModelConfiguration;
using EF_Sample05.DomainClasses.Models;

namespace EF_Sample05.DataLayer.Mappings
{
    public class CoachConfig : EntityTypeConfiguration<Coach>
    {
        public CoachConfig()
        {
            // For TPH
            this.Map(m => m.Requires(discriminator: "PersonType").HasValue(1));
        }
    }
}
```

```
using System.Data.Entity.ModelConfiguration;
using EF_Sample05.DomainClasses.Models;

namespace EF_Sample05.DataLayer.Mappings
{
    public class PlayerConfig : EntityTypeConfiguration<Player>
    {
        public PlayerConfig()
        {
            // For TPH
            this.Map(m => m.Requires(discriminator: "PersonType").HasValue(2));
        }
    }
}
```

در اینجا توسط متد Map، نام فیلد discriminator به PersonType تغییر کرده. همچنین چون مقدار پیش فرض تعیین شده توسط متد HasValue عددی است، نوع این فیلد در سمت بانک اطلاعاتی به int null تغییر می‌کند.

ب) Table per Type یا TPT

در حالت TPT، به ازای هر کلاس موجود در سلسله مراتب تعیین شده، یک جدول در سمت بانک اطلاعاتی تشکیل می‌گردد.
در جداول متناظر با Sub classes، تنها همان فیلدهایی وجود خواهند داشت که در کلاس‌های هم نام وجود دارد و فیلدهای کلاس پایه در آن‌ها ذکر نخواهد گردید. همچنین این جداول دارای یک Primary key نیز خواهند بود (که دقیقا همان کلید اصلی جدول پایه است که به آن Shared primary key هم گفته می‌شود). این کلید اصلی، به عنوان کلید خارجی اشاره کننده به کلاس یا جدول پایه نیز تنظیم می‌گردد:



شکل ۴

برای تنظیم این نوع ارث بری، تنها کافی است ویژگی Table را بر روی Sub classes قرار داد:

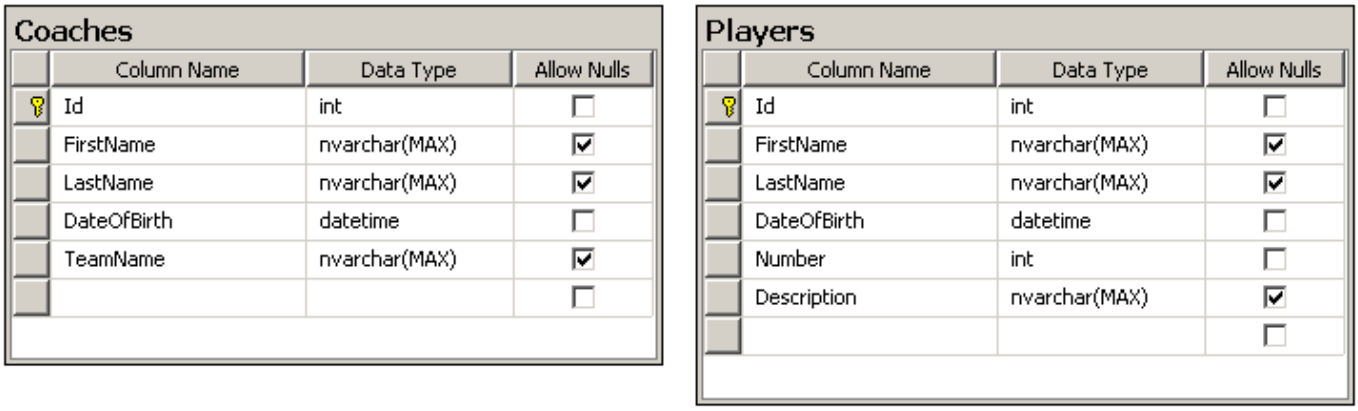
```
using System.ComponentModel.DataAnnotations;

namespace EF_Sample05.DomainClasses.Models
{
    [Table("Coaches")]
    public class Coach : Person
    {
        public string TeamName { set; get; }
    }
}
```

```
using System.ComponentModel.DataAnnotations;

namespace EF_Sample05.DomainClasses.Models
{
    [Table("Players")]
    public class Player : Person
    {
        public int Number { get; set; }
        public string Description { get; set; }
    }
}
```

یا اگر حالت Fluent API را ترجیح می‌دهید، همانطور که در قسمت‌های قبل نیز ذکر شد، معادل ویژگی Table در اینجا، متد ToTable است.

ج) Table per Concrete type یا TPC

در تعاریف ارث بری که تاکنون بررسی کردیم، مرسوم است کلاس پایه را از نوع abstract تعریف کنند. به این ترتیب هدف اصلی، Sub classes تعریف شده خواهند بود؛ چون نمی‌توان مستقیما وهله‌ای را از کلاس abstract تعریف شده ایجاد کرد.
در حالت TPC، به ازای هر sub class غیر abstract، یک جدول ایجاد می‌شود. هر جدول نیز حاوی فیلدهای کلاس پایه می‌باشد (برخلاف حالت TPT که جداول متناظر با کلاس‌های مشتق شده، تنها حاوی همان خواص و فیلدهای کلاس‌های متناظر بودند و نه بیشتر). به این ترتیب عملا جداول تشکیل شده در بانک اطلاعاتی، از وجود ارث بری در سمت کدهای ما بی‌خبر خواهند بود.



شکل ۵

برای پیاده سازی TPC نیاز است از Fluent API استفاده شود:

```
using System.ComponentModel.DataAnnotations;
using System.Data.Entity.ModelConfiguration;
using EF_Sample05.DomainClasses.Models;

namespace EF_Sample05.DataLayer.Mappings
{
    public class PersonConfig : EntityTypeConfiguration<Person>
    {
        public PersonConfig()
        {
            // for TPC
            this.Property(x => x.Id).HasDatabaseGeneratedOption(DatabaseGeneratedOption.None);
        }
    }
}
```

```
using System.Data.Entity.ModelConfiguration;
using EF_Sample05.DomainClasses.Models;

namespace EF_Sample05.DataLayer.Mappings
{
    public class CoachConfig : EntityTypeConfiguration<Coach>
    {
        public CoachConfig()
        {
            // For TPH
            //this.Map(m => m.Requires(discriminator: "PersonType").HasValue(1));

            // for TPT
            //this.ToTable("Coaches");

            //for TPC
            this.Map(m =>
            {
                m.MapInheritedProperties();
                m.ToTable("Coaches");
            });
```

```
        }
    }
}
```

```
using System.Data.Entity.ModelConfiguration;
using EF_Sample05.DomainClasses.Models;

namespace EF_Sample05.DataLayer.Mappings
{
    public class PlayerConfig : EntityTypeConfiguration<Player>
    {
        public PlayerConfig()
        {
            // For TPH
            //this.Map(m => m.Requires(discriminator: "PersonType").HasValue(2));

            // for TPT
            //this.ToTable("Players");

            //for TPC
            this.Map(m =>
                     {
                         m.MapInheritedProperties();
                         m.ToTable("Players");
                     });
        }
    }
}
```

ابتدا نوع فیلد Id از حالت Identity خارج شده است. این مورد جهت کار با TPC ضروری است در غیراینصورت EF هنگام ثبت، به مشکل بر می‌خورد، از این لحاظ که برای دو شیء، به یک Id خواهد رسید و امکان ثبت را نخواهد داد. بنابراین در یک چنین حالتی استفاده از نوع Guid برای تعریف primary key شاید بهتر باشد. بدیهی است در این حالت باید Id را به صورت دستی مقدار دهی نمود.

در ادامه توسط متد MapInheritedProperties، به همان مقصود لحاظ کردن تمام فیلدهای ارث بری شده در جدول حاصل، خواهیم رسید. همچنین نام جداول متناظر نیز ذکر گردیده است.

سؤال : از این بین، بهتر است از کدامیک استفاده شود؟

- برای حالت‌های ساده از TPH استفاده کنید. برای مثال یک بانک اطلاعاتی قدیمی دارید که هر جدول آن 200 تا یا شاید بیشتر فیلد دارد! امکان تغییر طراحی آن هم وجود ندارد. برای اینکه بتوان به حس بهتری حین کارکردن با این نوع سیستم‌های قدیمی رسید، می‌شود از ترکیب TPH و ComplexTypes (که در قسمت‌های قبل در مورد آن بحث شد) برای مدیریت بهتر این نوع جداول در سمت کدهای برنامه استفاده کرد.

- اگر علاقمند به استفاده از روابط پلی‌مرفیک هستید (برای مثال در کلاسی دیگر، ارجاعی به کلاس پایه Person وجود دارد) و sub classes دارای تعداد فیلدهای کمی هستند، از TPH استفاده کنید.

- اگر تعداد فیلدهای sub classes زیاد است و بسیار بیشتر است از کلاس پایه، از روش TPT استفاده کنید.

- اگر عمق ارث بری و تعداد سطوح تعریف شده بالا است، بهتر است از TPC استفاده کنید. حالت TPT از join استفاده می‌کند و حالت TPC از union برای تشکیل کوئری‌ها کمک خواهد گرفت

عنوان: EF Code First #10
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۲۳ ۱۷:۵۴:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Entity framework

حین کار با ORMهای پیشرفته، ویژگی‌های جالب توجهی در اختیار برنامه نویس‌ها قرار می‌گیرد که در زمان استفاده از کلاس‌های متداول SQLHelper از آن‌ها خبری نیست؛ مانند:

الف) Deferred execution

ب) Lazy loading

ج) Eager loading

نحوه بررسی SQL نهایی تولیدی توسط EF

برای توضیح موارد فوق، نیاز به مشاهده خروجی SQL نهایی حاصل از ORM است و همچنین شمارش تعداد بار رفت و برگشت به بانک اطلاعاتی. بهترین ابزاری را که برای این منظور می‌توان پیشنهاد داد، برنامه EF Profiler است. برای دریافت آن می‌توانید به این آدرس مراجعه کنید: (^) و (^)

پس از وارد کردن نام و آدرس ایمیل، یک مجوز یک ماهه آزمایشی، به آدرس ایمیل شما ارسال خواهد شد.
زمانیکه این فایل را در ابتدای اجرای برنامه به آن معرفی می‌کنید، محل ذخیره سازی نهایی آن جهت بازبینی بعدی، مسیر MyUserName\Local Settings\Application Data\EntityFramework Profiler خواهد بود.

استفاده از این برنامه هم بسیار ساده است:
الف) در برنامه خود، ارجاعی را به اسمبلی HibernatingRhinos.Profiler.Appender.dll که در پوشه برنامه EFProf موجود است، اضافه کنید.
ب) در نقطه آغاز برنامه، متد زیر را فراخوانی نمائید:

```
HibernatingRhinos.Profiler.Appender.EntityFramework.EntityFrameworkProfiler.Initialize();
```

نقطه آغاز برنامه می‌تواند متد Application_Start برنامه‌های وب، در متد Program.Main برنامه‌های ویندوزی کنسول و WinForms و در سازنده کلاس App برنامه‌های WPF باشد.
ج) برنامه EFProf را اجرا کنید.

مزایای استفاده از این برنامه
1) وابسته به بانک اطلاعاتی مورد استفاده نیست. (برخلاف برای مثال برنامه معروف SQL Server Profiler که فقط به همراه SQL Server ارائه می‌شود)
2) خروجی SQL نمایش داده شده را فرمت کرده و به همراه Syntax highlighting نیز هست.
3) کار این برنامه صرفا به لاگ کردن SQL تولیدی خلاصه نمی‌شود. یک سری از Best practices را نیز به شما گوشزد می‌کند. بنابراین اگر نیاز دارید سیستم خود را بر اساس دیدگاه یک متخصص بررسی کنید (یک Code review ارزشمند)، این ابزار می‌تواند بسیار مفید باشد.
4) می‌تواند کوئری‌های سنگین و سبک را به خوبی تشخیص داده و گزارشات آماری جالبی را به شما ارائه دهد.
5) می‌تواند دقیقا مشخص کند، کوئری را که مشاهده می‌کنید از طریق کدام متد در کدام کلاس صادر شده است و دقیقا از چه سطری.
6) امکان گروه بندی خودکار کوئری‌های صادر شده را بر اساس DbContext مورد استفاده به همراه دارد.
و ...

استفاده از این برنامه حین کار با EF «الزامی» است! (البته نسخه‌های NH و سایر ORMهای دیگر آن نیز موجود است و این مباحث در مورد تمام ORMهای پیشرفته صادق است)
مدام باید بررسی کرد که صفحه جاری چه تعداد کوئری را به بانک اطلاعاتی ارسال کرده و به چه نحوی. همچنین آیا می‌توان با اعمال اصلاحاتی، این وضع را بهبود بخشید. بنابراین عدم استفاده از این برنامه حین کار با ORMs، همانند راه رفتن در خواب است! ممکن است تصور کنید برنامه دارد به خوبی کار می‌کند اما ... در پشت صحنه فقط صفحه جاری برنامه، 100 کوئری را به بانک اطلاعاتی ارسال کرده، در حالیکه شما تنها نیاز به یک کوئری داشته‌اید.

کلاس‌های مدل مثال جاری

کلاس‌های مدل مثال جاری از یک دپارتمان که دارای تعدادی کارمند می‌باشد، تشکیل شده است. ضمنا هر کارمند تنها در یک دپارتمان می‌تواند مشغول به کار باشد و رابطه many-to-many نیست :

```
using System.Collections.Generic;

namespace EF_Sample06.Models
{
    public class Department
    {
        public int DepartmentId { get; set; }
        public string Name { get; set; }

        //Creates Employee navigation property for Lazy Loading (1:many)
        public virtual ICollection<Employee> Employees { get; set; }
    }
}
```

```
namespace EF_Sample06.Models
{
    public class Employee
    {
        public int EmployeeId { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }

        //Creates Department navigation property for Lazy Loading
        public virtual Department Department { get; set; }
    }
}
```

نگاشت دستی این کلاس‌ها هم ضرورتی ندارد، زیرا قراردادهای توکار EF Code first را رعایت کرده و EF در اینجا به سادگی می‌تواند primary key و روابط one-to-many را بر اساس navigation properties تعریف شده، تشخیص دهد.

در اینجا کلاس Context برنامه به شرح زیر است:

```
using System.Data.Entity;
using EF_Sample06.Models;

namespace EF_Sample06.DataLayer
{
    public class Sample06Context : DbContext
    {
        public DbSet<Department> Departments { set; get; }
        public DbSet<Employee> Employees { set; get; }
    }
}
```

و تنظیمات ابتدایی نحوه به روز رسانی و آغاز بانک اطلاعاتی نیز مطابق کدهای زیر می‌باشد:

```
using System.Collections.Generic;
using System.Data.Entity.Migrations;
using EF_Sample06.Models;

namespace EF_Sample06.DataLayer
{
    public class Configuration : DbMigrationsConfiguration<Sample06Context>
    {
        public Configuration()
        {
            AutomaticMigrationsEnabled = true;
            AutomaticMigrationDataLossAllowed = true;
        }

        protected override void Seed(Sample06Context context)
        {
            var employee1 = new Employee { FirstName = "f name1", LastName = "l name1" };
            var employee2 = new Employee { FirstName = "f name2", LastName = "l name2" };
            var employee3 = new Employee { FirstName = "f name3", LastName = "l name3" };
            var employee4 = new Employee { FirstName = "f name4", LastName = "l name4" };

            var dept1 = new Department { Name = "dept 1", Employees = new List<Employee> { employee1,
employee2 } };
            var dept2 = new Department { Name = "dept 2", Employees = new List<Employee> { employee3 }
};
            var dept3 = new Department { Name = "dept 3", Employees = new List<Employee> { employee4 }
};

            context.Departments.Add(dept1);
            context.Departments.Add(dept2);
            context.Departments.Add(dept3);
            base.Seed(context);
        }
    }
}
```

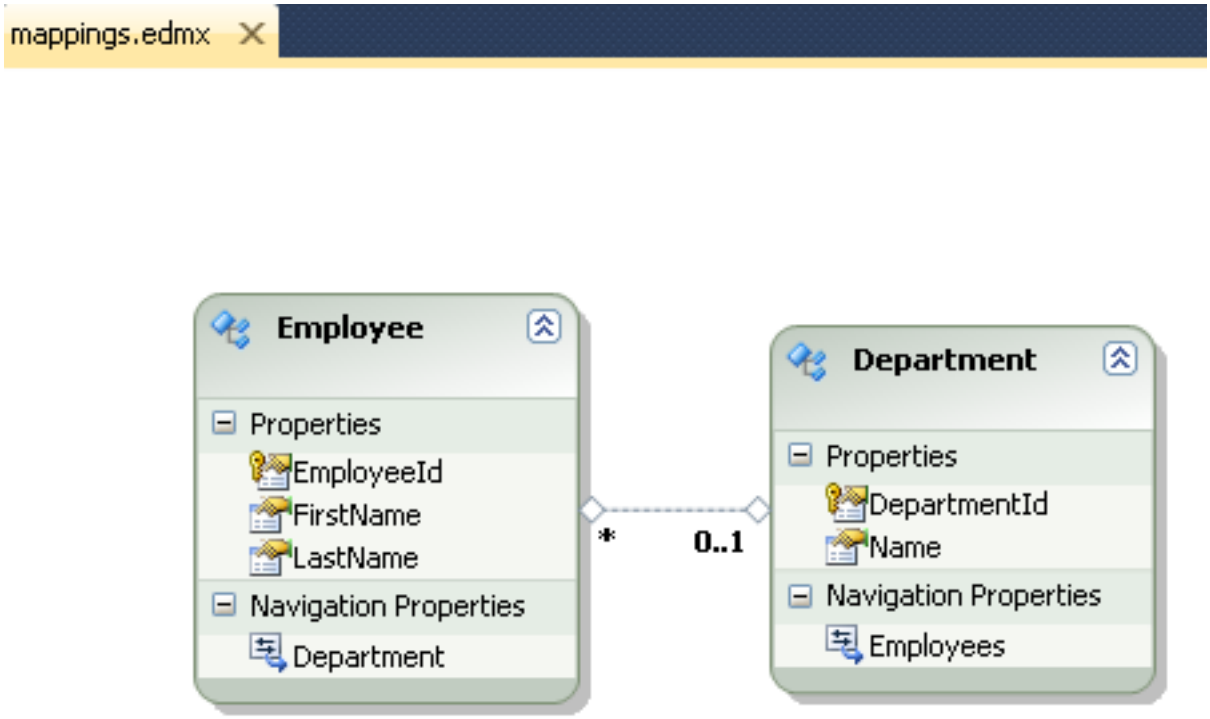
نکته: تهیه خروجی XML از نگاشت‌های خودکار تهیه شده

اگر علاقمند باشید که پشت صحنه نگاشت‌های خودکار EF Code first را در یک فایل XML جهت بررسی بیشتر ذخیره کنید، می‌توان از متد کمکی زیر استفاده کرد:

```
void ExportMappings(DbContext context, string edmxFile)
{
     var settings = new XmlWriterSettings { Indent = true };
     using (XmlWriter writer = XmlWriter.Create(edmxFile, settings))
     {
         System.Data.Entity.Infrastructure.EdmxWriter.WriteEdmx(context, writer);
     }
}
```

بهتر است پسوند فایل XML تولیدی را edmx قید کنید تا بتوان آن‌را با دوبار کلیک بر روی فایل، در ویژوال استودیو نیز مشاهده کرد:

```
using (var db = new Sample06Context())
{
     ExportMappings(db, "mappings.edmx");
}
```



شکل ۱

الف) بررسی Deferred execution یا بارگذاری به تاخیر افتاده

برای توضیح مفهوم Deferred loading/execution بهترین مثالی را که می‌توان ارائه داد، صفحات جستجوی ترکیبی در برنامه‌ها است. برای مثال یک صفحه جستجو را طراحی کرده‌اید که حاوی دو تکست باکس دریافت FirstName و LastName کاربر است. کنار هر کدام از این تکست باکس‌ها نیز یک چک‌باکس قرار دارد. به عبارتی کاربر می‌تواند جستجویی ترکیبی را در اینجا انجام دهد. نحوه پیاده سازی صحیح این نوع مثال‌ها در EF Code first به چه نحوی است؟

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using EF_Sample06.DataLayer;
using EF_Sample06.Models;

namespace EF_Sample06
{
    class Program
    {
        static IList<Employee> FindEmployees(string fName, string lName, bool byName, bool byLName)
        {
            using (var db = new Sample06Context())
            {
                IQueryable<Employee> query = db.Employees.AsQueryable();

                if (byLName)
                {
                    query = query.Where(x => x.LastName == lName);
                }

                if (byName)
                {
                    query = query.Where(x => x.FirstName == fName);
                }
```

```
                return query.ToList();
            }
        }

        static void Main(string[] args)
        {
            // note: remove this line if you received : create database is not supported by this
provider.
            HibernatingRhinos.Profiler.Appender.EntityFramework.EntityFrameworkProfiler.Initialize();

            Database.SetInitializer(new MigrateDatabaseToLatestVersion<Sample06Context,
Configuration>());

            var list = FindEmployees("f name1", "l name1", true, true);
            foreach (var item in list)
            {
                Console.WriteLine(item.FirstName);
            }
        }
    }
}
```

نحوه صحیح این نوع پیاده سازی ترکیبی را در متد FindEmployees مشاهده می‌کنید. نکته مهم آن، استفاده از نوع IQueryable و متد AsQueryable است و امکان ترکیب کوئری‌ها با هم.

به نظر شما با فراخوانی متد FindEmployees به نحو زیر که هر دو شرط آن توسط کاربر انتخاب شده است، چه تعداد کوئری به بانک اطلاعاتی ارسال می‌شود؟

```
var list = FindEmployees("f name1", "l name1", true, true);
```

شاید پاسخ دهید که سه بار : یکبار در متد db.Employees.AsQueryable و دوبار هم در حین ورود به بدنه شرط‌های یاد شده و اینجا است که کسانی که قبلا با رویه‌های ذخیره شده کار کرده باشند، شروع به فریاد و فغان می‌کنند که ما قبلا این مسایل رو با یک SP در یک رفت و برگشت مدیریت می‌کردیم!

پاسخ صحیح: «فقط یکبار»! آن‌هم تنها در زمان فراخوانی متد ToList و نه قبل از آن.

برای اثبات این مدعا نیاز است به خروجی SQL لاگ شده توسط EF Profiler مراجعه کرد:

```
SELECT [Extent1].[EmployeeId]              AS [EmployeeId],
       [Extent1].[FirstName]               AS [FirstName],
       [Extent1].[LastName]                AS [LastName],
       [Extent1].[Department_DepartmentId] AS [Department_DepartmentId]
FROM   [dbo].[Employees] AS [Extent1]
WHERE  ([Extent1].[LastName] = 'l name1' /* @p__linq__0 */)
       AND ([Extent1].[FirstName] = 'f name1' /* @p__linq__1 */)
```

IQueryable قلب LINQ است و تنها بیانگر یک عبارت (expression) از رکوردهایی می‌باشد که مد نظر شما است و نه بیشتر. برای مثال زمانیکه یک IQueryable را همانند مثال فوق فیلتر می‌کنید، هنوز چیزی از بانک اطلاعاتی یا منبع داده‌ای دریافت نشده است. هنوز هیچ اتفاقی رخ نداده است و هنوز رفت و برگشتی به منبع داده‌ای صورت نگرفته است. به آن باید به شکل یک expression builder نگاه کرد و نه لیستی از اشیاء فیلتر شده‌ی ما. به این مفهوم، deferred execution (اجرای به تاخیر افتاده) نیز گفته می‌شود.

کوئری LINQ شما تنها زمانی بر روی بانک اطلاعاتی اجرا می‌شود که کاری بر روی آن صورت گیرد مانند فراخوانی متد ToList، فراخوانی متد First یا FirstOrDefault و امثال آن. تا پیش از این فقط به شکل یک عبارت در برنامه وجود دارد و نه بیشتر.

اطلاعات بیشتر: « تفاوت بین IQueryable و IEnumerable در حین کار با ORMs »

ب) بررسی Lazy Loading یا واکشی در صورت نیاز

در مطلب جاری اگر به کلاس‌های مدل برنامه دقت کنید، تعدادی از خواص به صورت virtual تعریف شده‌اند. چرا؟
تعریف یک خاصیت به صورت virtual، پایه و اساس lazy loading است و به کمک آن، تا به اطلاعات شیءایی نیاز نباشد، وهله سازی نخواهد شد. به این ترتیب می‌توان به کارآیی بیشتری در حین کار با ORMs رسید. برای مثال در کلاس‌های فوق، اگر تنها نیاز به دریافت نام یک دپارتمان هست، نباید حین وهله سازی از شیء دپارتمان، شیء لیست کارمندان مرتبط با آن نیز وهله سازی شده و از بانک اطلاعاتی دریافت شوند. به این وهله سازی با تاخیر، lazy loading گفته می‌شود.
Lazy loading پیاده سازی ساده‌ای نداشته و مبتنی است بر بکارگیری AOP frameworks یا کتابخانه‌هایی که امکان تشکیل اشیاء Proxy پویا را در پشت صحنه فراهم می‌کنند. علت virtual تعریف کردن خواص رابط نیز به همین مساله بر می‌گردد، تا این نوع کتابخانه‌ها بتوانند در نحوه تعریف اینگونه خواص virtual در زمان اجرا، در پشت صحنه دخل و تصرف کنند. البته حین استفاده از EF یا انواع و اقسام ORMs دیگر با این نوع پیچیدگی‌ها روبرو نخواهیم شد و تشکیل اشیاء Proxy در پشت صحنه انجام می‌شوند.

یک مثال: قصد داریم اولین دپارتمان ثبت شده در حین آغاز برنامه را یافته و سپس لیست کارمندان آن‌را نمایش دهیم:

```
using (var db = new Sample06Context())
{
    var dept1 = db.Departments.Find(1);
    if (dept1 != null)
    {
        Console.WriteLine(dept1.Name);
        foreach (var item in dept1.Employees)
        {
             Console.WriteLine(item.FirstName);
        }
    }
}
```

رفتار یک ORM جهت تعیین اینکه آیا نیاز است برای دریافت اطلاعات بین جداول Join صورت گیرد یا خیر، واکشی حریصانه و غیرحریصانه را مشخص می‌سازد.
در حالت واکشی حریصانه به ORM خواهیم گفت که لطفا جهت دریافت اطلاعات فیلدهای جداول مختلف، از همان ابتدای کار در پشت صحنه، Join های لازم را تدارک ببین. در حالت واکشی غیرحریصانه به ORM خواهیم گفت به هیچ عنوان حق نداری Join ایی را تشکیل دهی. هر زمانی که نیاز به اطلاعات فیلدی از جدولی دیگر بود باید به صورت مستقیم به آن مراجعه کرده و آن مقدار را دریافت کنی.
به صورت خلاصه برنامه نویس در حین کار با ORM های پیشرفته نیازی نیست Join بنویسد. تنها باید ORM را طوری تنظیم کند که آیا اینکار را حتما خودش در پشت صحنه انجام دهد (واکشی حریصانه)، یا اینکه خیر، به هیچ عنوان SQL های تولیدی در پشت صحنه نباید حاوی Join باشند (lazy loading).

در مثال فوق به صورت خودکار دو کوئری به بانک اطلاعاتی ارسال می‌گردد:

```
SELECT [Limit1].[DepartmentId] AS [DepartmentId],
       [Limit1].[Name]         AS [Name]
FROM   (SELECT TOP (2) [Extent1].[DepartmentId] AS [DepartmentId],
                       [Extent1].[Name]         AS [Name]
        FROM   [dbo].[Departments] AS [Extent1]
        WHERE  [Extent1].[DepartmentId] = 1 /* @p0 */) AS [Limit1]


SELECT [Extent1].[EmployeeId]              AS [EmployeeId],
       [Extent1].[FirstName]               AS [FirstName],
       [Extent1].[LastName]                AS [LastName],
       [Extent1].[Department_DepartmentId] AS [Department_DepartmentId]
FROM   [dbo].[Employees] AS [Extent1]
```

```
WHERE  ([Extent1].[Department_DepartmentId] IS NOT NULL)
       AND ([Extent1].[Department_DepartmentId] = 1 /* @EntityKeyValue1 */)
```

یکبار زمانیکه قرار است اطلاعات دپارتمان یک (db.Departments.Find) دریافت شود. تا این لحظه خبری از جدول Employees نیست. چون lazy loading فعال است و فقط اطلاعاتی را که نیاز داشته‌ایم فراهم کرده است.
زمانیکه برنامه به حلقه می‌رسد، نیاز است اطلاعات dept1.Employees را دریافت کند. در اینجا است که کوئری دوم، به بانک اطلاعاتی صادر خواهد شد (بارگذاری در صورت نیاز).

ج) بررسی Eager Loading یا واکشی حریصانه

حالت lazy loading بسیار جذاب به نظر می‌رسد؛ برای مثال می‌توان خواص حجیم یک جدول را به جدول مرتبط دیگری منتقل کرد. مثلا فیلدهای متنی طولانی یا اطلاعات باینری فایل‌های ذخیره شده، تصاویر و امثال آن. به این ترتیب تا زمانیکه نیازی به اینگونه اطلاعات نباشد، lazy loading از بارگذاری آن‌ها جلوگیری کرده و سبب افزایش کارآیی برنامه می‌شود.
اما ... همین lazy loading در صورت استفاده نا آگاهانه می‌تواند سرور بانک اطلاعاتی را در یک برنامه چندکاربره از پا درآورد! نیازی هم نیست تا شخصی به سایت شما حمله کند. مهاجم اصلی همان برنامه نویس کم اطلاع است!
اینبار مثال زیر را درنظر بگیرید که بجای دریافت اطلاعات یک شخص، مثلا قصد داریم، اطلاعات کلیه دپارتمان‌ها را توسط یک Grid نمایش دهیم (فرقی نمی‌کند برنامه وب یا ویندوز باشد؛ اصول یکی است):

```
using (var db = new Sample06Context())
{
      foreach (var dept in db.Departments)
      {
           Console.WriteLine(dept.Name);
           foreach (var item in dept.Employees)
           {
                Console.WriteLine(item.FirstName);
            }
        }
}
```

یک نکته: اگر سعی کنیم کد فوق را اجرا کنیم به خطای زیر برخواهیم خورد:

```
There is already an open DataReader associated with this Command which must be closed first
```

برای رفع این مشکل نیاز است گزینه MultipleActiveResultSets=True را به کانکشن استرینگ اضافه کرد:

```
<connectionStrings>
    <clear/>
    <add
       name="Sample06Context"
       connectionString="Data Source=(local);Initial Catalog=testdb2012;Integrated Security =
true;MultipleActiveResultSets=True;"
       providerName="System.Data.SqlClient"
      />
</connectionStrings>
```

سؤال: به نظر شما در دو حلقه تو در توی فوق چندبار رفت و برگشت به بانک اطلاعاتی صورت می‌گیرد؟ با توجه به اینکه در متد Seed ذکر شده در ابتدای مطلب، تعداد رکوردها مشخص است.

پاسخ: 7 بار!

Object context #15

Statements | Object context Usage

Short SQL	Row Count	Duration	Alerts
SELECT ... FROM [dbo].[Departments] AS [Extent1]	6	27 ms / 891 ms	
SELECT ... FROM [dbo].[Employees] AS [Extent1] WHERE...	2	63 ms / 94 ms	
SELECT ... FROM [dbo].[Employees] AS [Extent1] WHERE...	1	44 ms / 47 ms	
SELECT ... FROM [dbo].[Employees] AS [Extent1] WHERE...	1	294 ms / 297 ms	
SELECT ... FROM [dbo].[Employees] AS [Extent1] WHERE...	2	38 ms / 47 ms	
SELECT ... FROM [dbo].[Employees] AS [Extent1] WHERE...	1	162 ms / 172 ms	
SELECT ... FROM [dbo].[Employees] AS [Extent1] WHERE...	1	209 ms / 203 ms	

Details | Alerts | Stack Trace

```
1 SELECT [Extent1].[EmployeeId]              AS [EmployeeId],
2        [Extent1].[FirstName]               AS [FirstName],
3        [Extent1].[LastName]                AS [LastName],
4        [Extent1].[Department_DepartmentId] AS [Department_DepartmentId]
5 FROM   [dbo].[Employees] AS [Extent1]
6 WHERE  ([Extent1].[Department_DepartmentId] IS NOT NULL)
7        AND ([Extent1].[Department_DepartmentId] = 21 /* @EntityKeyValue1
```

Param	Value
@EntityK	21

شکل ۲

و اینجا است که عنوان شد استفاده از EF Profiler در حین توسعه برنامه‌های مبتنی بر ORM «الزامی» است! اگر از این نکته اطلاعی نداشتید، بهتر است یکبار تمام صفحات گزارش‌گیری برنامه‌های خود را که حاوی یک Grid هستند، توسط EF Profiler بررسی کنید. اگر در این برنامه پیغام خطای n+1 select را دریافت کردید، یعنی در حال استفاده ناصحیح از امکانات lazy loading می‌باشید.

آیا می‌توان این وضعیت را بهبود بخشید؟ زمانیکه کار ما گزارشگیری از اطلاعات با تعداد رکوردهای بالا است، استفاده ناصحیح از ویژگی Lazy loading می‌تواند به شدت کارآیی بانک اطلاعاتی را پایین بیاورد. برای حل این مساله در زمان‌های قدیم (!) بین جداول join می‌نوشتند؛ الان چطور؟
در EF متدی به نام Include جهت Eager loading اطلاعات موجودیت‌های مرتبط به هم درنظر گرفته شده است که در پشت صحنه همینکار را انجام می‌دهد:

```
using (var db = new Sample06Context())
{
      foreach (var dept in db.Departments.Include(x => x.Employees))
      {
           Console.WriteLine(dept.Name);
           foreach (var item in dept.Employees)
           {
              Console.WriteLine(item.FirstName);
           }
       }
}
```

همانطور که ملاحظه می‌کنید اینبار به کمک متد Include، نسبت به واکشی حریصانه Employees اقدام کرده‌ایم. اکنون اگر برنامه را اجرا کنیم، فقط یک رفت و برگشت به بانک اطلاعاتی انجام خواهد شد و کار Join نویسی به صورت خودکار توسط EF مدیریت می‌گردد:

```
SELECT [Project1].[DepartmentId]            AS [DepartmentId],
       [Project1].[Name]                    AS [Name],
       [Project1].[C1]                      AS [C1],
       [Project1].[EmployeeId]              AS [EmployeeId],
       [Project1].[FirstName]               AS [FirstName],
       [Project1].[LastName]                AS [LastName],
       [Project1].[Department_DepartmentId] AS [Department_DepartmentId]
FROM   (SELECT [Extent1].[DepartmentId]            AS [DepartmentId],
               [Extent1].[Name]                    AS [Name],
               [Extent2].[EmployeeId]              AS [EmployeeId],
               [Extent2].[FirstName]               AS [FirstName],
               [Extent2].[LastName]                AS [LastName],
               [Extent2].[Department_DepartmentId] AS [Department_DepartmentId],
               CASE
                 WHEN ([Extent2].[EmployeeId] IS NULL) THEN CAST(NULL AS int)
                 ELSE 1
               END                                 AS [C1]
        FROM   [dbo].[Departments] AS [Extent1]
               LEFT OUTER JOIN [dbo].[Employees] AS [Extent2]
                 ON [Extent1].[DepartmentId] = [Extent2].[Department_DepartmentId]) AS [Project1]
ORDER  BY [Project1].[DepartmentId] ASC,
          [Project1].[C1] ASC
```

متد Include در نگارش‌های اخیر EF پیشرفت کرده است و همانند مثال فوق، امکان کار با lambda expressions را جهت تعریف خواص مورد نظر به صورت strongly typed ارائه می‌دهد. در نگارش‌های قبلی این متد، تنها امکان استفاده از رشته‌ها برای معرفی خواص وجود داشت.

همچنین توسط متد Include امکان eager loading چندین سطح با هم نیز وجود دارد؛ مثلا x.Employees.Kids و همانند آن.

چند نکته در مورد نحوه خاموش کردن Lazy loading

امکان خاموش کردن Lazy loading در تمام کلاس‌های برنامه با تنظیم خاصیت Configuration.LazyLoadingEnabled کلاس Context برنامه به نحو زیر میسر است:

```
public class Sample06Context : DbContext
{
        public Sample06Context()
        {
            this.Configuration.LazyLoadingEnabled = false;
        }
```

یا اگر تنها در مورد یک کلاس نیاز است این خاموش سازی صورت گیرد، کلمه کلیدی virtual را حذف کنید. برای مثال با نوشتن public ICollection<Employee> Employees بجای public virtual ICollection<Employee> Employees در اولین بار وهله سازی کلاس دپارتمان، لیست کارمندان آن به نال تنظیم می‌شود. البته در این حالت null object pattern را نیز فراموش نکنید (وهله سازی پیش فرض Employees در سازنده کلاس):

```
public class Department
{
     public int DepartmentId { get; set; }
```

```
    public string Name { get; set; }

    public  ICollection<Employee> Employees { get; set; }
    public Department()
    {
        Employees = new HashSet<Employee>();
    }
}
```

به این ترتیب به خطای null reference object بر نخواهیم خورد. همچنین وهله سازی، با مقدار دهی لیست دریافتی از بانک اطلاعاتی متفاوت است. در اینجا نیز باید از متد Include استفاده کرد.

بنابراین در صورت خاموش کردن lazy loading، حتما نیاز است از متد Include استفاده شود. اگرlazy loading فعال است، جهت تبدیل آن به eager loading از متد Include استفاده کنید (اما اجباری نیست).

عنوان: EF Code First #11
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۲۴ ۲۲:۵۱:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Entity framework

استفاده از الگوی Repository اضافی در EF Code first؛ آری یا خیر؟!

اگر در ویژوال استودیو، اشاره‌گر ماوس را بر روی تعریف DbContext قرار دهیم، راهنمای زیر ظاهر می‌شود:

```
A DbContext instance represents a combination of the Unit Of Work and Repository patterns such that
it can be used to query from a database and group together changes that will then be written back to
the store as a unit.  DbContext is conceptually similar to ObjectContext.
```

در اینجا تیم EF صراحتا عنوان می‌کند که DbContext در EF Code first همان الگوی Unit Of Work را پیاده سازی کرده و در داخل کلاس مشتق شده از آن، DbSetها همان Repositories هستند (فقط نام‌ها تغییر کرده‌اند؛ اصول یکی است).
به عبارت دیگر با نام بردن صریح از این الگوها، مقصود زیر را دنبال می‌کنند:
لطفا بر روی این لایه Abstraction ایی که ما تهیه دیده‌ایم، یک لایه Abstraction دیگر را ایجاد نکنید!
«لایه Abstraction دیگر» یعنی پیاده سازی الگوهای Unit Of Work و Repository جدید، برفراز الگوهای Unit Of Work و Repository توکار موجود!
کار اضافه‌ای که در بسیاری از سایت‌ها مشاهده می‌شود و ... متاسفانه اکثر آن‌ها هم اشتباه هستند! در ذیل روش‌های تشخیص پیاده سازی‌های نادرست الگوی Repository را بر خواهیم شمرد:

1) قرار دادن متد Save تغییرات نهایی انجام شده، در داخل کلاس Repository

متد Save باید داخل کلاس Unit of work تعریف شود نه داخل کلاس Repository. دقیقا همان کاری که در EF Code first به درستی انجام شده. متد SaveChanges توسط DbContext ارائه می‌شود. علت هم این است که در زمان Save ممکن است با چندین Entity و چندین جدول مشغول به کار باشیم. حاصل یک تراکنش، باید نهایتا ذخیره شود نه اینکه هر کدام از این‌ها، تراکنش خاص خودشان را داشته باشند.

2) نداشتن درکی از الگوی Unit of work

به Unit of work به شکل یک تراکنش نگاه کنید. در داخل آن با انواع و اقسام موجودیت‌ها از کلاس‌ها و جداول مختلف کار شده و حاصل عملیات، به بانک اطلاعاتی اعمال می‌گردد. پیاده سازی‌های اشتباه الگوی Repository، تمام امکانات را در داخل همان کلاس Repository قرار می‌دهند؛ که اشتباه است. این نوع کلاس‌ها فقط برای کار با یک Entity بهینه شده‌اند؛ در حالیکه در دنیای واقعی، اطلاعات ممکن است از دو Entity مختلف دریافت و نتیجه محاسبات مفروضی به Entity سوم اعمال شود. تمام این عملیات یک تراکنش را تشکیل می‌دهد، نه اینکه هر کدام، تراکنش مجزای خود را داشته باشند.

3) وهله سازی از DbContext به صورت مستقیم داخل کلاس Repository

4) Dispose اشیاء DbContext داخل کلاس Repository

هر بار وهله سازی DbContext مساوی است با باز شدن یک اتصال به بانک اطلاعاتی و همچنین از آنجائیکه راهنمای ذکر شده فوق را در مورد DbContext مطالعه نکرده‌اند، زمانیکه در یک متد با سه وهله از سه Repository موجودیت‌های مختلف کار می‌کنید، سه تراکنش و سه اتصال مختلف به بانک اطلاعاتی گشوده شده است. این مورد ذاتا اشتباه است و سربار بالایی را نیز به همراه دارد.
ضمن اینکه بستن DbContext در یک Repository، امکان اعمال کوئری‌های بعدی LINQ را غیرممکن می‌کند. به ظاهر یک شیء IQueryable در اختیار داریم که می‌توان بر روی آن انواع و اقسام کوئری‌های LINQ را تعریف کرد اما ... در اینجا با LINQ to Objects که بر روی اطلاعات موجود در حافظه کار می‌کند سر و کار نداریم. اتصال به بانک اطلاعاتی با بستن DbContext قطع شده، بنابراین کوئری LINQ بعدی شما کار نخواهد کرد.
همچنین در EF نمی‌توان یک Entity را از یک Context به Context دیگری ارسال کرد. در پیاده سازی صحیح الگوی Repository (دقیقا همان چیزی که در EF Code first به صورت توکار وجود دارد)، Context باید بین Repositories که در اینجا فقط نامش

DbSet تعریف شده، به اشتراک گذاشته شود. علت هم این است که EF از Context برای ردیابی تغییرات انجام شده بر روی موجودیت‌ها استفاده می‌کند (همان سطح اول کش که در قسمت‌های قبل به آن اشاره شد). اگر به ازای هر Repository یکبار وهله سازی DbContext انجام شود، هر کدام کش جداگانه خاص خود را خواهند داشت.
5) عدم امکان استفاده از تنها یک DbConetext به ازای یک Http Request
هنگامیکه وهله سازی DbContext به داخل یک Repository منتقل می‌شود و الگوی واحد کار رعایت نمی‌گردد، امکان به اشتراک گذاری آن بین Repositoryهای تعریف شده وجود نخواهد داشت. این مساله در برنامه‌های وب سبب کاهش کارآیی می‌گردد (باز و بسته شدن بیش از حد اتصال به بانک اطلاعاتی در حالیکه می‌شد تمام این عملیات را با یک DbContext انجام داد).

نمونه‌ای از این پیاده سازی اشتباه را در اینجا می‌توانید پیدا کنید. متاسفانه شبیه به همین پیاده سازی، در پروژه MVC Scaffolding نیز بکارگرفته شده است.

چرا تعریف لایه دیگری بر روی لایه Abstraction موجود در EF Code first اشتباه است؟

یکی از دلایلی که حین تعریف الگوی Repository دوم بر روی لایه موجود عنوان می‌شود، این است:
« **به این ترتیب به سادگی می‌توان ORM مورد استفاده را تغییر داد** » چون پیاده سازی استفاده از ORM، در پشت این لایه مخفی شده و ما هر زمان که بخواهیم به ORM دیگری کوچ کنیم، فقط کافی است این لایه را تغییر دهیم و نه کل برنامه را.
ولی سؤال این است که هرچند این مساله از هزار فرسنگ بالاتر درست است، اما واقعا تابحال دیده‌اید که پروژه‌ای را با یک ORM شروع کنند و بعد سوئیچ کنند به ORM دیگری؟!
ضمنا برای اینکه واقعا لایه اضافی پیاده سازی شده انتقال پذیر باشد، شما باید کاملا دست و پای ORM موجود را بریده و توانایی‌های در دسترس آن را به سطح نازلی کاهش دهید تا پیاده سازی شما قابل انتقال باشد. برای مثال یک سری از قابلیت‌های پیشرفته و بسیار جالب در NH هست که در EF نیست و برعکس. آیا واقعا می‌توان به همین سادگی ORM مورد استفاده را تغییر داد؟ فقط در یک حالت این امر میسر است: از قابلیت‌های پیشرفته ابزار موجود استفاده نکنیم و از آن در سطحی بسیار ساده و ابتدایی کمک بگیریم تا از قابلیت‌های مشترک بین ORMهای موجود استفاده شود.
ضمن اینکه مباحث نگاشت کلاس‌ها به جداول را چکار خواهید کرد؟ EF راه و روش خاص خودش را دارد، NH چندین و چند روش خاص خودش را دارد! این‌ها به این سادگی قابل انتقال نیستند که شخصی عنوان کند: «هر زمان که علاقمند بودیم، ORM مورد استفاده را می‌شود عوض کرد!»

دلیل دومی که برای تهیه لایه اضافه‌تری بر روی DbContext عنوان می‌کنند این است:
« **با استفاده از الگوی Repository نوشتن آزمون‌های واحد ساده‌تر می‌شود** ». زمانیکه برنامه بر اساس Interfaceها کار می‌کند می‌توان آن‌ها را بجای اشاره به بانک اطلاعاتی، به نمونه‌ای موجود در حافظه، در زمان آزمون تغییر داد.
این مورد در حالت کلی درست است اما .... نه در مورد بانک‌های اطلاعاتی!
زمانیکه در یک آزمون واحد، پیاده سازی جدیدی از الگوی Interface مخزن ما تهیه می‌شود و اینبار بجای بانک اطلاعاتی با یک سری شیء قرارگرفته در حافظه سروکار داریم، آیا موارد زیر را هم می‌توان به سادگی آزمایش کرد؟
ارتباطات بین جداول‌را، cascade delete، فیلدهای identity، فیلدهای unique، کلیدهای ترکیبی، نوع‌های خاص تعریف شده در بانک اطلاعاتی و مسایلی از این دست.
پاسخ: خیر! تغییر انجام شده، سبب کار برنامه با اطلاعات موجود در حافظه خواهد شد، یعنی LINQ to Objects.
شما در حالت استفاده از LINQ to Objects آزادی عمل فوق العاده‌ای دارید. می‌توانید از انواع و اقسام متدها حین تهیه کوئری‌های LINQ استفاده کنید که هیچکدام معادلی در بانک اطلاعاتی نداشته و ... به ظاهر آزمون واحد شما پاس می‌شود؛ اما در عمل بر روی یک بانک اطلاعاتی واقعی کار نخواهد کرد.
البته شاید شخصی عنوان که بله می‌شود تمام این‌ها نیازمندی‌ها را در حالت کار با اشیاء درون حافظه هم پیاده سازی کرد ولی ... در نهایت پیاده سازی آن بسیار پیچیده و در حد پیاده سازی یک بانک اطلاعاتی واقعی خواهد شد که واقعا ضرورتی ندارد.

و پاسخ صحیح در اینجا و این مساله خاص این است:
لطفا در حین کار با بانک‌های اطلاعاتی مباحث mocking را فراموش کنید. بجای SQL Server، رشته اتصالی و تنظیمات برنامه را به SQL Server CE تغییر داده و آزمایشات خود را انجام دهید. پس از پایان کار هم بانک اطلاعاتی را delete کنید. به این نوع آزمون‌ها اصطلاحا integration tests گفته می‌شود. لازم است برنامه با یک بانک اطلاعاتی واقعی تست شود و نه یک سری شیء ساده

قرار گرفته در حافظه که هیچ قیدی همانند شرایط کار با یک بانک اطلاعاتی واقعی، بر روی آن‌ها اعمال نمی‌شود.
ضمنا باید درنظر داشت بانک‌های اطلاعاتی که تنها در حافظه کار کنند نیز وجود دارند. برای مثال SQLite حالت کار کردن صرفا در حافظه را پشتیبانی می‌کند. زمانیکه آزمون واحد شروع می‌شود، یک بانک اطلاعاتی واقعی را در حافظه تشکیل داده و پس از پایان کار هم ... اثری از این بانک اطلاعاتی باقی نخواهد ماند و برای این نوع کارها بسیار سریع است.

نتیجه گیری:
حین استفاده از EF code first، الگوی واحد کار، همان DbContext است و الگوی مخزن، همان DbSetها. ضرورتی به ایجاد یک لایه محافظ اضافی بر روی این‌ها وجود ندارد.
در اینجا بهتر است یک لایه اضافی را به نام مثلا Service ایجاد کرد و تمام اعمال کار با EF را به آن منتقل نمود. سپس در قسمت‌های مختلف برنامه می‌توان از متدهای این لایه استفاده کرد. به عبارتی در فایل‌های Code behind برنامه شما نباید کدهای EF مشاهده شوند. یا در کنترلرهای MVC نیز به همین ترتیب. این‌ها مصرف کننده نهایی لایه سرویس ایجاد شده خواهند بود.
همچنین بجای نوشتن آزمون‌های واحد، به Integration tests سوئیچ کنید تا بتوان برنامه را در شرایط کار با یک بانک اطلاعاتی واقعی تست کرد.

برای مطالعه بیشتر:
Abstracting your ORM is a futile exercise
Repository is the new Singleton
Night of the living Repositories
Architecting in the pit of doom: The evils of the repository abstraction layer
?Is Repository old skool
?EF Code First: Where’s My Repository
Generic Repository With EF 4.1 what is the point

عنوان: EF Code First #12
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۲۶ ۱۰:۵۸:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Entity framework

پیاده سازی الگوی Context Per Request در برنامه‌های مبتنی بر EF Code first

در طراحی برنامه‌های چند لایه مبتنی بر EF مرسوم نیست که در هر کلاس و متدی که قرار است از امکانات آن استفاده کند، یکبار DbContext و کلاس مشتق شده از آن وهله سازی شوند؛ به این ترتیب امکان انجام امور مختلف در طی یک تراکنش از بین می‌رود. برای حل این مشکل الگویی مطرح شده است به نام Session/Context Per Request و یا به اشتراک گذاری یک Unit of work در لایه‌های مختلف برنامه در طی یک درخواست، که در ادامه یک پیاده سازی آن‌را با هم مرور خواهیم کرد.
البته این سشن با سشن ASP.NET یکی نیست. در NHibernate معادل DbContextایی که در اینجا ملاحظه می‌کنید، Session نام دارد.

اهمیت بکارگیری الگوی Unit of work و به اشتراک گذاری آن در طی یک درخواست

در الگوی واحد کار یا همان DbContext در اینجا، تمام درخواست‌های رسیده به آن، در صف قرار گرفته و تمام آن‌ها در پایان کار، به بانک اطلاعاتی اعمال می‌شوند. برای مثال زمانیکه شیءایی را به یک وهله از DbContext اضافه/حذف می‌کنیم، یا در ادامه مقدار خاصیتی را تغییر می‌دهیم، هیچکدام از این تغییرات تا زمانیکه متد SaveChanges فراخوانی نشود، به بانک اطلاعاتی اعمال نخواهند شد. این مساله مزایای زیر را به همراه خواهد داشت:

الف) کارآیی بهتر
در اینجا از یک کانکشن باز شده، حداکثر استفاده صورت می‌گیرد. چندین و چند عملیات در طی یک batch به بانک اطلاعاتی اعمال می‌گردند؛ بجای اینکه برای اعمال هرکدام، یکبار اتصال جداگانه‌ای به بانک اطلاعاتی باز شود.

ب) بررسی مسایل همزمانی
استفاده از یک الگوی واحد کار، امکان بررسی خودکار تمام تغییرات انجام شده بر روی یک موجودیت را در متدها و لایه‌های مختلف میسر کرده و به این ترتیب مسایل مرتبط با ConcurrencyMode عنوان شده در قسمت‌های قبل به نحو بهتری قابل مدیریت خواهند بود.

ج) استفاده صحیح از تراکنش‌ها
الگوی واحد کار به صورت خودکار از تراکنش‌ها استفاده می‌کند. اگر در حین فراخوانی متد SaveChanges مشکلی رخ دهد، کل عملیات Rollback خواهد شد و تغییری در بانک اطلاعاتی رخ نخواهد داد. بنابراین استفاده از یک تراکنش در حین چند عملیات ناشی از لایه‌های مختلف برنامه، منطقی‌تر است تا اینکه هر کدام، در تراکنشی جدا مشغول به کار باشند.

کلاس‌های مدل مثال جاری

در مثالی که در این قسمت بررسی خواهیم کرد، از کلاس‌های مدل گروه محصولات کمک گرفته شده است:

```
using System.Collections.Generic;

namespace EF_Sample07.DomainClasses
{
    public class Category
    {
        public int Id { get; set; }
        public virtual string Name { get; set; }
```

```
        public virtual string Title { get; set; }

        public virtual ICollection<Product> Products { get; set; }
    }
}
```

```
using System.ComponentModel.DataAnnotations;

namespace EF_Sample07.DomainClasses
{
    public class Product
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }

        [ForeignKey("CategoryId")]
        public virtual Category Category { get; set; }
        public int CategoryId { get; set; }
    }
}
```

در کلاس Product، یک خاصیت اضافی به نام CategoryId اضافه شده است که توسط ویژگی ForeignKey، به عنوان کلید خارجی جدول معرفی خواهد شد. از این خاصیت در برنامه‌های ASP.NET برای مقدار دهی یک کلید خارجی توسط یک DropDownList پر شده با لیست گروه‌ها، استفاده خواهیم کرد.

پیاده سازی الگوی واحد کار

همانطور که در قسمت قبل نیز ذکر شد، DbContext در EF Code first بر اساس الگوی واحد کار تهیه شده است، اما برای به اشتراک گذاشتن آن بین لایه‌های مختلف برنامه نیاز است یک لایه انتزاعی را برای آن تهیه کنیم، تا بتوان آن‌را به صورت خودکار توسط کتابخانه‌های Dependency Injection یا به اختصار DI در زمان نیاز به استفاده از آن، به کلاس‌های استفاده کننده تزریق کنیم. کتابخانه‌ی DI ایی که در این قسمت مورد استفاده قرار می‌گیرد، کتابخانه معروف StructureMap است. برای دریافت آن می‌توانید از Nuget استفاده کنید؛ یا از صفحه اصلی آن در Github : (^).
اینترفیس پایه الگوی واحد کار ما به شرح زیر است:

```
using System.Data.Entity;
using System;

namespace EF_Sample07.DataLayer.Context
{
    public interface IUnitOfWork
    {
        IDbSet<TEntity> Set<TEntity>() where TEntity : class;
        int SaveChanges();
    }
}
```

برای استفاده اولیه آن، تنها تغییری که در برنامه حاصل می‌شود به نحو زیر است:

```
using System.Data.Entity;
using EF_Sample07.DomainClasses;

namespace EF_Sample07.DataLayer.Context
```

```
{
    public class Sample07Context : DbContext, IUnitOfWork
    {
        public DbSet<Category> Categories { set; get; }
        public DbSet<Product> Products { set; get; }

        #region IUnitOfWork Members
        public new IDbSet<TEntity> Set<TEntity>() where TEntity : class
        {
            return base.Set<TEntity>();
        }
        #endregion
    }
}
```

توضیحات:

با کلاس Context در قسمت‌های قبل آشنا شده‌ایم. در اینجا به معرفی کلاس‌هایی خواهیم پرداخت که در معرض دید EF Code first قرار خواهند گرفت.

DbSetها هم معرف الگوی Repository هستند. کلاس Sample07Context، معرفی الگوی واحد کار یا Unit of work برنامه است.

برای اینکه بتوانیم تعاریف کلاس‌های سرویس برنامه را مستقل از تعریف کلاس Sample07Context کنیم، یک اینترفیس جدید را به نام IUnitOfWork به برنامه اضافه کرده‌ایم.

در اینجا کلاس Sample07Context پیاده سازی کننده اینترفیس IUnitOfWork خواهد بود (اولین تغییر).

دومین تغییر هم استفاده از متد base.Set می‌باشد. به این ترتیب به سادگی می‌توان به DbSetهای مختلف در حین کار با IUnitOfWork دسترسی پیدا کرد. به عبارتی ضرورتی ندارد به ازای تک تک DbSetها یکبار خاصیت جدیدی را به اینترفیس IUnitOfWork اضافه کرد. به کمک استفاده از امکانات Generics مهیا، اینبار

```
uow.Set<Product>
```

معادل همان db.Products سابق است؛ در حالتیکه از Sample07Context به صورت مستقیم استفاده شود.

همچنین نیازی به پیاده سازی متد SaveChanges نیست؛ زیرا پیاده سازی آن در کلاس DbContext قرار دارد.

استفاده از الگوی واحد کار در کلاس‌های لایه سرویس برنامه

```
using EF_Sample07.DomainClasses;
using System.Collections.Generic;

namespace EF_Sample07.ServiceLayer
{
    public interface ICategoryService
    {
        void AddNewCategory(Category category);
        IList<Category> GetAllCategories();
    }
}
```

```
using EF_Sample07.DomainClasses;
using System.Collections.Generic;

namespace EF_Sample07.ServiceLayer
{
    public interface IProductService
    {
        void AddNewProduct(Product product);
        IList<Product> GetAllProducts();
    }
}
```

لایه سرویس برنامه را با دو اینترفیس جدید شروع می‌کنیم. هدف از این اینترفیس‌ها، ارائه پیاده سازی‌های متفاوت، به ازای ORMهای مختلف است. برای مثال در کلاس‌های زیر که نام آن‌ها با Ef شروع شده است، پیاده سازی خاص Ef Code first را تدارک خواهیم دید. این پیاده سازی، قابل انتقال به سایر ORMها نیست چون نه پیاده سازی یکسانی را از مباحث LINQ ارائه می‌دهند و نه متدهای الحاقی همانندی را به همراه دارند و نه اینکه مباحث نگاشت کلاس‌های آن‌ها به جداول مختلف یکی است:

```
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using EF_Sample07.DataLayer.Context;
using EF_Sample07.DomainClasses;

namespace EF_Sample07.ServiceLayer
{
    public class EfCategoryService : ICategoryService
    {
        IUnitOfWork _uow;
        IDbSet<Category> _categories;
        public EfCategoryService(IUnitOfWork uow)
        {
            _uow = uow;
            _categories = _uow.Set<Category>();
        }

        public void AddNewCategory(Category category)
        {
            _categories.Add(category);
        }

        public IList<Category> GetAllCategories()
        {
            return _categories.ToList();
        }
    }
}
```

```
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using EF_Sample07.DataLayer.Context;
using EF_Sample07.DomainClasses;

namespace EF_Sample07.ServiceLayer
{
    public class EfProductService : IProductService
    {
        IUnitOfWork _uow;
        IDbSet<Product> _products;
        public EfProductService(IUnitOfWork uow)
        {
            _uow = uow;
            _products = _uow.Set<Product>();
        }

        public void AddNewProduct(Product product)
        {
            _products.Add(product);
        }

        public IList<Product> GetAllProducts()
        {
            return _products.Include(x => x.Category).ToList();
        }
    }
}
```

توضیحات:

همانطور که ملاحظه می‌کنید در هیچکدام از کلاس‌های سرویس برنامه، وهله سازی مستقیمی از الگوی واحد کار وجود ندارد. این لایه از برنامه اصلا نمی‌داند که کلاسی به نام Sample07Context وجود خارجی دارد یا خیر.

همچنین لایه اضافی دیگری را به نام Repository جهت مخفی سازی سازوکار EF به برنامه اضافه نکرده‌ایم. این لایه شاید در نگاه اول برنامه را مستقل از ORM جلوه دهد اما در عمل قابل انتقال نیست و سبب تحمیل سربار اضافی بی موردی به برنامه می‌شود؛ ORMها ویژگی‌های یکسانی را ارائه نمی‌دهند. حتی در حالت استفاده از LINQ، پیاده سازی‌های یکسانی را به همراه ندارند.

بنابراین اگر قرار است برنامه مستقل از ORM کار کند، نیاز است لایه استفاده کننده از سرویس برنامه، با دو اینترفیس IProductService و ICategoryService کار کند و نه به صورت مستقیم با پیاده سازی آن‌ها. به این ترتیب هر زمان که لازم شد، فقط باید پیاده سازی‌های کلاس‌های سرویس را تغییر داد؛ باز هم برنامه نهایی بدون نیاز به تغییری کار خواهد کرد.

تا اینجا به معماری پیچیده‌ای نرسیده‌ایم و اصطلاحا over-engineering صورت نگرفته است. یک اینترفیس بسیار ساده IUnitOfWork به برنامه اضافه شده؛ در ادامه این اینترفیس به کلاس‌های سرویس برنامه تزریق شده است (تزریق وابستگی در سازنده کلاس). کلاس‌های سرویس ما «می‌دانند» که EF وجود خارجی دارد و سعی نکرده‌ایم توسط لایه اضافی دیگری آن‌را مخفی کنیم. شیوه کار با IDbSet تعریف شده دقیقا همانند روال متداولی است که با EF Code first کار می‌شود و بسیار طبیعی جلوه می‌کند.

استفاده از الگوی واحد کار و کلاس‌های سرویس تهیه شده در یک برنامه کنسول ویندوزی

در ادامه برای وهله سازی اینترفیس‌های سرویس و واحد کار برنامه، از کتابخانه StructureMap که یاد شد، استفاده خواهیم کرد. بنابراین، تمام برنامه‌های نهایی ارائه شده در این قسمت، ارجاعی را به اسمبلی StructureMap.dll نیاز خواهند داشت.

کدهای برنامه کنسول مثال جاری را در ادامه ملاحظه خواهید کرد:

```
using System.Collections.Generic;
using System.Data.Entity;
using EF_Sample07.DataLayer.Context;
using EF_Sample07.DomainClasses;
using EF_Sample07.ServiceLayer;
using StructureMap;

namespace EF_Sample07
{
    class Program
    {
        static void Main(string[] args)
        {
            Database.SetInitializer(new MigrateDatabaseToLatestVersion<Sample07Context,
Configuration>());

            HibernatingRhinos.Profiler.Appender.EntityFramework.EntityFrameworkProfiler.Initialize();

            ObjectFactory.Initialize(x =>
            {
                x.For<IUnitOfWork>().CacheBy(InstanceScope.Hybrid).Use<Sample07Context>();
                x.For<ICategoryService>().Use<EfCategoryService>();
            });

            var uow = ObjectFactory.GetInstance<IUnitOfWork>();
            var categoryService = ObjectFactory.GetInstance<ICategoryService>();

            var product1 = new Product { Name = "P100", Price = 100 };
            var product2 = new Product { Name = "P200", Price = 200 };
            var category1 = new Category
            {
                Name = "Cat100",
                Title = "Title100",
                Products = new List<Product> { product1, product2 }
            };
            categoryService.AddNewCategory(category1);

            uow.SaveChanges();
        }
    }
}
```

در اینجا بیشتر هدف، معرفی نحوه استفاده از StructureMap است.
ابتدا توسط متد ObjectFactory.Initialize مشخص می‌کنیم که اگر برنامه نیاز به اینترفیس IUnitOfWork داشت، لطفا کلاس Sample07Context را وهله سازی کرده و مورد استفاده قرار بده. اگر ICategoryService مورد استفاده قرار گرفت، وهله مورد نظر باید از کلاس EfCategoryService تامین شود.
توسط ObjectFactory.GetInstance نیز می‌توان به وهله‌ای از این کلاس‌ها دست یافت و نهایتا با فراخوانی uow.SaveChanges می‌توان اطلاعات را ذخیره کرد.

چند نکته:
- به کمک کتابخانه StructureMap، تزریق IUnitOfWork به سازنده کلاس EfCategoryService به صورت خودکار انجام می‌شود. اگر به کدهای فوق دقت کنید ما فقط با اینترفیس‌ها مشغول به کار هستیم، اما وهله‌سازی‌ها در پشت صحنه انجام می‌شود.
- حین معرفی IUnitOfWork از متد CacheBy با پارامتر InstanceScope.Hybrid استفاده شده است. این enum مقادیر زیر را می‌تواند بپذیرد:

```
public enum InstanceScope
{
        PerRequest = 0,
        Singleton = 1,
        ThreadLocal = 2,
        HttpContext = 3,
        Hybrid = 4,
        HttpSession = 5,
        HybridHttpSession = 6,
        Unique = 7,
        Transient = 8,
}
```

برای مثال اگر در برنامه‌ای نیاز داشتید یک کلاس به صورت Singleton عمل کند، فقط کافی است نحوه کش شدن آن‌را تغییر دهید.
حالت PerRequest در برنامه‌های وب کاربرد دارد (و حالت پیش فرض است). با انتخاب آن وهله سازی کلاس مورد نظر به ازای هر درخواست رسیده انجام خواهد شد.
در حالت ThreadLocal، به ازای هر Thread، وهله‌ای متفاوت در اختیار مصرف کننده قرار می‌گیرد.
با انتخاب حالت HttpContext، به ازای هر HttpContext ایجاد شده، کلاس معرفی شده یکبار وهله سازی می‌گردد.
حالت Hybrid ترکیبی است از حالت‌های HttpContext و ThreadLocal. اگر برنامه وب بود، از HttpContext استفاده خواهد کرد در غیراینصورت به ThreadLocal سوئیچ می‌کند.

استفاده از الگوی واحد کار و کلاس‌های سرویس تهیه شده در یک برنامه ASP.NET MVC

یک برنامه خالی ASP.NET MVC را آغاز کنید. سپس یک HomeController جدید را نیز به آن اضافه نمائید و کدهای آن‌را مطابق اطلاعات زیر تغییر دهید:

```
using System.Web.Mvc;
using EF_Sample07.DomainClasses;
using EF_Sample07.ServiceLayer;
using EF_Sample07.DataLayer.Context;
using System.Collections.Generic;

namespace EF_Sample07.MvcAppSample.Controllers
{
    public class HomeController : Controller
    {
        IProductService _productService;
```

```
        ICategoryService _categoryService;
        IUnitOfWork _uow;
        public HomeController(IUnitOfWork uow, IProductService productService, ICategoryService
categoryService)
        {
            _productService = productService;
            _categoryService = categoryService;
            _uow = uow;
        }

        [HttpGet]
        public ActionResult Index()
        {
            var list = _productService.GetAllProducts();
            return View(list);
        }

        [HttpGet]
        public ActionResult Create()
        {
            ViewBag.CategoriesList = new SelectList(_categoryService.GetAllCategories(), "Id", "Name");
            return View();
        }

        [HttpPost]
        public ActionResult Create(Product product)
        {
            if (this.ModelState.IsValid)
            {
                _productService.AddNewProduct(product);
                _uow.SaveChanges();
            }
            return RedirectToAction("Index");
        }

        [HttpGet]
        public ActionResult CreateCategory()
        {
            return View();
        }

        [HttpPost]
        public ActionResult CreateCategory(Category category)
        {
            if (this.ModelState.IsValid)
            {
                _categoryService.AddNewCategory(category);
                _uow.SaveChanges();
            }
            return RedirectToAction("Index");
        }
    }
}
```

نکته مهم این کنترلر، تزریق وابستگی‌ها در سازنده کلاس کنترلر است؛ به این ترتیب کنترلر جاری نمی‌داند که با کدام پیاده سازی خاصی از این اینترفیس‌ها قرار است کار کند.

اگر برنامه را به همین نحو اجرا کنیم، موتور ASP.NET MVC ایراد خواهد گرفت که یک کنترلر باید دارای سازنده‌ای بدون پارامتر باشد تا من بتوانم به صورت خودکار وهله‌ای از آن‌را ایجاد کنم. برای رفع این مشکل از کتابخانه StructureMap برای تزریق خودکار وابستگی‌ها کمک خواهیم گرفت:

```
using System;
using System.Data.Entity;
using System.Web.Mvc;
using System.Web.Routing;
using EF_Sample07.DataLayer.Context;
using EF_Sample07.ServiceLayer;
using StructureMap;

namespace EF_Sample07.MvcAppSample
{
    // Note: For instructions on enabling IIS6 or IIS7 classic mode,
    // visit http://go.microsoft.com/?LinkId=9394801
```

```
    public class MvcApplication : System.Web.HttpApplication
    {
        public static void RegisterGlobalFilters(GlobalFilterCollection filters)
        {
            filters.Add(new HandleErrorAttribute());
        }

        public static void RegisterRoutes(RouteCollection routes)
        {
            routes.IgnoreRoute("{resource}.axd/{*pathInfo}");

            routes.MapRoute(
                "Default", // Route name
                "{controller}/{action}/{id}", // URL with parameters
                new { controller = "Home", action = "Index", id = UrlParameter.Optional } // Parameter
defaults
            );

        }

        protected void Application_Start()
        {
            Database.SetInitializer(new MigrateDatabaseToLatestVersion<Sample07Context,
Configuration>());
            HibernatingRhinos.Profiler.Appender.EntityFramework.EntityFrameworkProfiler.Initialize();

            AreaRegistration.RegisterAllAreas();

            RegisterGlobalFilters(GlobalFilters.Filters);
            RegisterRoutes(RouteTable.Routes);
            initStructureMap();
        }

        private static void initStructureMap()
        {
            ObjectFactory.Initialize(x =>
            {
                x.For<IUnitOfWork>().HttpContextScoped().Use(() => new Sample07Context());
                x.ForRequestedType<ICategoryService>().TheDefaultIsConcreteType<EfCategoryService>();
                x.ForRequestedType<IProductService>().TheDefaultIsConcreteType<EfProductService>();
            });
            //Set current Controller factory as StructureMapControllerFactory
            ControllerBuilder.Current.SetControllerFactory(new StructureMapControllerFactory());
        }

        protected void Application_EndRequest(object sender, EventArgs e)
        {
            ObjectFactory.ReleaseAndDisposeAllHttpScopedObjects();
        }
    }

    public class StructureMapControllerFactory : DefaultControllerFactory
    {
        protected override IController GetControllerInstance(RequestContext requestContext, Type
controllerType)
        {
            return ObjectFactory.GetInstance(controllerType) as Controller;
        }
    }
}
```

توضیحات:

کدهای فوق متعلق به کلاس Global.asax.cs هستند. در اینجا در متد Application_Start، متد initStructureMap فراخوانی شده است.

با پیاده سازی ObjectFactory.Initialize در کدهای برنامه کنسول معرفی شده آشنا شدیم. اینبار فقط حالت کش شدن کلاس Context برنامه را HttpContextScoped قرار داده‌ایم تا به ازای هر درخواست رسیده یک بار الگوی واحد کار وهله سازی شود.

نکته مهمی که در اینجا اضافه شده‌است، استفاده از متد ControllerBuilder.Current.SetControllerFactory می‌باشد. این متد نیاز به وهله‌ای از نوع DefaultControllerFactory دارد که نمونه‌ای از آن‌را در کلاس StructureMapControllerFactory مشاهده می‌کنید. به این ترتیب در زمان وهله سازی خودکار یک کنترلر، اینبار StructureMap وارد عمل شده و وابستگی‌های برنامه را مطابق تعاریف ObjectFactory.Initialize ذکر شده، به سازنده کلاس کنترلر تزریق می‌کند.

همچنین در متد Application_EndRequest با فراخوانی ObjectFactory.ReleaseAndDisposeAllHttpScopedObjects از نشتی اتصالات به بانک اطلاعاتی جلوگیری خواهیم کرد. چون وهله الگوی کار برنامه HttpScoped تعریف شده، در پایان یک درخواست به صورت خودکار توسط StructureMap پاکسازی می‌شود و به نشتی منابع نخواهیم رسید.

استفاده از الگوی واحد کار و کلاس‌های سرویس تهیه شده در یک برنامه ASP.NET Web forms

در یک برنامه ASP.NET Web forms نیز می‌توان این مباحث را پیاده سازی کرد:

```
using System;
using System.Data.Entity;
using EF_Sample07.DataLayer.Context;
using EF_Sample07.ServiceLayer;
using StructureMap;

namespace EF_Sample07.WebFormsAppSample
{
    public class Global : System.Web.HttpApplication
    {
        private static void initStructureMap()
        {
            ObjectFactory.Initialize(x =>
            {
                x.For<IUnitOfWork>().HttpContextScoped().Use(() => new Sample07Context());
                x.ForRequestedType<ICategoryService>().TheDefaultIsConcreteType<EfCategoryService>();
                x.ForRequestedType<IProductService>().TheDefaultIsConcreteType<EfProductService>();

                x.SetAllProperties(y=>
                {
                    y.OfType<IUnitOfWork>();
                    y.OfType<ICategoryService>();
                    y.OfType<IProductService>();
                });
            });
        }

        void Application_Start(object sender, EventArgs e)
        {
            Database.SetInitializer(new MigrateDatabaseToLatestVersion<Sample07Context,
Configuration>());
            HibernatingRhinos.Profiler.Appender.EntityFramework.EntityFrameworkProfiler.Initialize();

            initStructureMap();
        }

        void Application_EndRequest(object sender, EventArgs e)
        {
            ObjectFactory.ReleaseAndDisposeAllHttpScopedObjects();
        }
```

در اینجا کدهای کلاس Global.asax.cs را ملاحظه می‌کنید. توضیحات آن با قسمت ASP.NET MVC آنچنان تفاوتی ندارد و یکی است. البته منهای تعاریف SetAllProperties که جدید است و در ادامه به علت اضافه کردن آن‌ها خواهیم رسید.
در ASP.NET Web forms برخلاف ASP.NET MVC نیاز است کار وهله سازی اینترفیس‌ها را به صورت دستی انجام دهیم. برای این منظور و کاهش کدهای تکراری برنامه می‌توان یک کلاس پایه را به نحو زیر تعریف کرد:

```
using System.Web.UI;
using StructureMap;

namespace EF_Sample07.WebFormsAppSample
{
    public class BasePage : Page
    {
        public BasePage()
        {
            ObjectFactory.BuildUp(this);
        }
```

```
    }
}
```

سپس برای استفاده از آن خواهیم داشت:

```
using System;
using EF_Sample07.DataLayer.Context;
using EF_Sample07.DomainClasses;
using EF_Sample07.ServiceLayer;

namespace EF_Sample07.WebFormsAppSample
{
    public partial class AddProduct : BasePage
    {
        public IUnitOfWork UoW { set; get; }
        public IProductService ProductService { set; get; }
        public ICategoryService CategoryService { set; get; }

        protected void Page_Load(object sender, EventArgs e)
        {
            if (!IsPostBack)
            {
                bindToCategories();
            }
        }

        private void bindToCategories()
        {
            ddlCategories.DataTextField = "Name";
            ddlCategories.DataValueField = "Id";
            ddlCategories.DataSource = CategoryService.GetAllCategories();
            ddlCategories.DataBind();
        }

        protected void btnAdd_Click(object sender, EventArgs e)
        {
            var product = new Product
            {
                Name = txtName.Text,
                Price = int.Parse(txtPrice.Text),
                CategoryId = int.Parse(ddlCategories.SelectedItem.Value)
            };
            ProductService.AddNewProduct(product);
            UoW.SaveChanges();
            Response.Redirect("~/Default.aspx");
        }
    }
}
```

اینبار وابستگی‌های کلاس افزودن محصولات، به صورت خواصی عمومی تعریف شده‌اند. این خواص عمومی توسط متد SetAllProperties که در فایل global.asax.cs معرفی شدند، باید یکبار تعریف شوند (مهم!).
سپس اگر دقت کرده باشید، اینبار کلاس AddProduct از BasePage ما ارث بری کرده است. در سازند کلاس BasePage، با فراخوانی متد ObjectFactory.BuildUp، تزریق وابستگی‌ها به خواص عمومی کلاس جاری صورت می‌گیرد.
در ادامه نحوه استفاده از این اینترفیس‌ها را جهت مقدار دهی یک DropDownList یا ذخیره سازی اطلاعات یک محصول مشاهده می‌کنید. در اینجا نیز کار با اینترفیس‌ها انجام شده و کلاس جاری دقیقا نمی‌داند که با چه وهله‌ای مشغول به کار است. تنها در زمان اجرا است که توسط StructureMap ، به ازای هر اینترفیس معرفی شده، وهله‌ای مناسب بر اساس تعاریف فایل Global.asax.cs در اختیار برنامه قرار می‌گیرد.

کدهای کامل مثال‌های این سری را از آدرس زیر هم می‌توانید دریافت کنید: (^)

عنوان: EF Code First #13
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۲۷ ۱۰:۰۶:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Entity framework

استفاده مستقیم از عبارات SQL در EF Code first

طراحی اکثر ORMهای موجود به نحوی است که برنامه نهایی شما را مستقل از بانک اطلاعاتی کنند و این پروایدر نهایی است که معادل‌های صحیح بسیاری از توابع توکار بانک اطلاعاتی مورد استفاده را در اختیار EF قرار می‌دهد. برای مثال در یک بانک اطلاعاتی تابعی به نام substr تعریف شده، در بانک اطلاعاتی دیگری همین تابع substring نام دارد. اگر برنامه را به کمک کوئری‌های LINQ تهیه کنیم، نهایتا پروایدر نهایی مخصوص بانک اطلاعاتی مورد استفاده است که این معادل‌ها را در اختیار EF قرار می‌دهد و برنامه بدون مشکل کار خواهد کرد. اما یک سری از موارد شاید معادلی در سایر بانک‌های اطلاعاتی نداشته باشند؛ برای مثال رویه‌های ذخیره شده یا توابع تعریف شده توسط کاربر. امکان استفاده از یک چنین توانایی‌هایی نیز با اجرای مستقیم عبارات SQL در EF Code first پیش بینی شده و بدیهی است در این حالت برنامه به یک بانک اطلاعاتی خاص گره خواهد خورد؛ همچنین مزیت استفاده از کوئری‌های Strongly typed تحت نظر کامپایلر را نیز از دست خواهیم داد. به علاوه باید به یک سری مسایل امنیتی نیز دقت داشت که در ادامه بررسی خواهند شد.

کلاس‌های مدل مثال جاری

در مثال جاری قصد داریم نحوه استفاده از رویه‌های ذخیره شده و توابع تعریف شده توسط کاربر مخصوص SQL Server را بررسی کنیم. در اینجا کلاس‌های پزشک و بیماران او، کلاس‌های مدل برنامه را تشکیل می‌دهند:

```
using System.Collections.Generic;

namespace EF_Sample08.DomainClasses
{
    public class Doctor
    {
        public int Id { set; get; }
        public string Name { set; get; }

        public virtual ICollection<Patient> Patients { set; get; }
    }
}
```

```
namespace EF_Sample08.DomainClasses
{
    public class Patient
    {
        public int Id { set; get; }
        public string Name { set; get; }

        public virtual Doctor Doctor { set; get; }
    }
}
```

کلاس **Context** برنامه به نحو زیر تعریف شده:

```
using System.Data.Entity;
using EF_Sample08.DomainClasses;
```

```
namespace EF_Sample08.DataLayer.Context
{
    public class Sample08Context : DbContext
    {
        public DbSet<Doctor> Doctors { set; get; }
        public DbSet<Patient> Patients { set; get; }
    }
}
```

و اینبار کلاس **DbMigrationsConfiguration** تعریف شده اندکی با مثال‌های قبلی متفاوت است:

```
using System.Data.Entity.Migrations;
using EF_Sample08.DomainClasses;
using System.Collections.Generic;

namespace EF_Sample08.DataLayer.Context
{
    public class Configuration : DbMigrationsConfiguration<Sample08Context>
    {
        public Configuration()
        {
            AutomaticMigrationsEnabled = true;
            AutomaticMigrationDataLossAllowed = true;
        }

        protected override void Seed(Sample08Context context)
        {
            addData(context);
            addSP(context);
            addFn(context);
            base.Seed(context);
        }

        private static void addData(Sample08Context context)
        {
            var patient1 = new Patient { Name = "p1" };
            var patient2 = new Patient { Name = "p2" };
            var doctor1 = new Doctor { Name = "doc1", Patients = new List<Patient> { patient1, patient2
} };
            context.Doctors.Add(doctor1);
        }

        private static void addFn(Sample08Context context)
        {
            context.Database.ExecuteSqlCommand(
                @"IF  EXISTS (SELECT * FROM sys.objects WHERE object_id =
OBJECT_ID(N'[dbo].[FindDoctorPatientsCount]')
                  AND type in (N'FN', N'IF', N'TF', N'FS', N'FT'))
                     DROP FUNCTION [dbo].[FindDoctorPatientsCount]");
            context.Database.ExecuteSqlCommand(
                @"CREATE FUNCTION FindDoctorPatientsCount(@Doctor_Id INT)
                    RETURNS INT
                    BEGIN
                     RETURN
                     (
                         SELECT COUNT(*)
                         FROM   Patients
                         WHERE  Doctor_Id = @Doctor_Id
                     );
                    END");
        }

        private static void addSP(Sample08Context context)
        {
            context.Database.ExecuteSqlCommand(
                @"IF  EXISTS (SELECT * FROM sys.objects WHERE object_id =
OBJECT_ID(N'[dbo].[FindDoctorsStartWith]')
                  AND type in (N'P', N'PC'))
                       DROP PROCEDURE [dbo].[FindDoctorsStartWith]
                 ");
            context.Database.ExecuteSqlCommand(
                @"CREATE PROCEDURE FindDoctorsStartWith(@name NVARCHAR(400))
                  AS
                 SELECT *
                 FROM   Doctors
```

```
                  WHERE  [Name] LIKE @name + '%'");
        }
    }
}
```

در اینجا از متد Seed علاوه بر مقدار دهی اولیه جداول، برای تعریف یک رویه ذخیره شده به نام FindDoctorsStartWith و یک تابع سفارشی به نام FindDoctorPatientsCount نیز استفاده شده است. متد context.Database.ExecuteSqlCommand مستقیما یک عبارت SQL را بر روی بانک اطلاعاتی اجرا می‌کند.

در ادامه کدهای کامل برنامه نهایی را ملاحظه می‌کنید:

```
using System;
using System.Data;
using System.Data.Entity;
using System.Data.Objects.SqlClient;
using System.Data.SqlClient;
using System.Linq;
using EF_Sample08.DataLayer.Context;
using EF_Sample08.DomainClasses;

namespace EF_Sample08
{
    class Program
    {
        static void Main(string[] args)
        {
            Database.SetInitializer(new MigrateDatabaseToLatestVersion<Sample08Context,
Configuration>());

            using (var db = new Sample08Context())
            {
                runSp(db);
                runFn(db);
                usingSqlFunctions(db);
            }
        }

        private static void usingSqlFunctions(Sample08Context db)
        {
            var doctorsWithNumericNameList = db.Doctors.Where(x => SqlFunctions.IsNumeric(x.Name) ==
1).ToList();
            if (doctorsWithNumericNameList.Any())
            {
                //do something
            }
        }

        private static void runFn(Sample08Context db)
        {
            var doctorIdParameter = new SqlParameter
            {
                ParameterName = "@doctor_id",
                Value = 1,
                SqlDbType = SqlDbType.Int
            };
            var patientsCount = db.Database.SqlQuery<int>("select
dbo.FindDoctorPatientsCount(@doctor_id)", doctorIdParameter).FirstOrDefault();
            Console.WriteLine(patientsCount);
        }

        private static void runSp(Sample08Context db)
        {
            var nameParameter = new SqlParameter
            {
                ParameterName = "@name",
                Value = "doc",
                Direction = ParameterDirection.Input,
                SqlDbType = SqlDbType.NVarChar
            };
            var doctors = db.Database.SqlQuery<Doctor>("exec FindDoctorsStartWith @name",
nameParameter).ToList();
            if (doctors.Any())
            {
```

```
                foreach (var item in doctors)
                {
                    Console.WriteLine(item.Name);
                }
            }
        }
    }
}
```

توضیحات

همانطور که ملاحظه می‌کنید، برای اجرای مستقیم یک عبارت SQL صرفنظر از اینکه یک رویه ذخیره شده است یا یک تابع و یا یک کوئری معمولی، باید از متد db.Database.SqlQuery استفاده کرد. خروجی این متد از نوع IEnumerable است و این توانایی را دارد که رکوردهای بازگشت داده شده از بانک اطلاعاتی را به خواص یک کلاس به صورت خودکار نگاشت کند.
پارامتر اول متد db.Database.SqlQuery، عبارت SQL مورد نظر است. پارامتر دوم آن باید توسط وهله‌هایی از کلاس SqlParameter مقدار دهی شود. به کمک SqlParameter نام پارامتر مورد استفاده، مقدار و نوع آن مشخص می‌گردد. همچنین Direction آن نیز برای استفاده از رویه‌های ذخیره شده ویژه‌ای که دارای پارامتری از نوع out هستند درنظر گرفته شده است.

چند نکته

- در متد runSp فوق، متد الحاقی ToList را حذف کرده و برنامه را اجرا کنید. بلافاصله پیغام خطای «The SqlParameter is already contained by another SqlParameterCollection.» ظاهر خواهد شد. علت هم این است که با بکارگیری متد ToList، تمام عملیات یکبار انجام شده و نتیجه بازگشت داده می‌شود اما اگر به صورت مستقیم از خروجی IEnumerable آن استفاده کنیم، در حلقه foreach تعریف شده، ممکن است این فراخوانی چندبار انجام شود. به همین جهت ذکر متد ToList در اینجا ضروری است.

- عنوان شد که در اینجا باید به مسایل امنیتی دقت داشت. بدیهی است امکان نوشتن یک چنین کوئری‌هایی نیز وجود دارد:

```
db.Database.SqlQuery<Doctor>("exec FindDoctorsStartWith "+ txtName.Text, nameParameter).ToList()
```

در این حالت به ظاهر مشغول به استفاده از رویه‌های ذخیره شده‌ای هستیم که عنوان می‌شود در برابر حملات تزریق SQL در امان هستند، اما چون در کدهای ما به نحو ناصحیحی با جمع زدن رشته‌ها مقدار دهی شده است، برنامه و بانک اطلاعاتی دیگر در امان نخواهند بود. بنابراین در این حالت استفاده از پارامترها را نباید فراموش کرد.
زمانیکه از کوئری‌های LINQ استفاده می‌شود تمام این مسایل توسط EF مدیریت خواهد شد. اما اگر قصد دارید مستقیما عبارات SQL را فراخوانی کنید، تامین امنیت برنامه به عهده خودتان خواهد بود.

- در متد usingSqlFunctions از SqlFunctions.IsNumeric استفاده شده است. این مورد مختص به SQL Server است و امکان استفاده از توابع توکار ویژه SQL Server را در کوئری‌های LINQ برنامه فراهم می‌سازد. برای مثال متدالحاقی از پیش تعریف شده‌ای به نام IsNumeric به صورت مستقیم در دسترس نیست، اما به کمک کلاس SqlFunctions این تابع و بسیاری از توابع دیگر توکار SQL Server قابل استفاده خواهند بود.
اگر علاقمند هستید که لیست این توابع را مشاهده کنید، در ویژوال استودیو بر روی SqlFunctions کلیک راست کرده و گزینه Go to definition را انتخاب کنید.

عنوان: EF Code First #14
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۲۹ ۱۷:۱۷:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Entity framework

ردیابی تغییرات در EF Code first

EF از DbContext برای ذخیره اطلاعات مرتبط با تغییرات موجودیت‌های تحت کنترل خود کمک می‌گیرد. این نوع اطلاعات توسط Change Tracker API جهت بررسی وضعیت فعلی یک شیء، مقادیر اصلی و مقادیر تغییر کرده آن در دسترس هستند. همچنین در اینجا امکان بارگذاری مجدد اطلاعات موجودیت‌ها از بانک اطلاعاتی جهت اطمینان از به روز بودن آن‌ها تدارک دیده شده است. ساده‌ترین روش دستیابی به این اطلاعات، استفاده از متد context.Entry می‌باشد که یک وهله از موجودیتی خاص را دریافت کرده و سپس به کمک خاصیت State خروجی آن، وضعیت‌هایی مانند Unchanged یا Modified را می‌توان به دست آورد. علاوه بر آن خروجی متد context.Entry، دارای خواصی مانند CurrentValues و OriginalValues نیز می‌باشد. OriginalValues شامل مقادیر خواص موجودیت درست در لحظه اولین بارگذاری در DbContext برنامه است. CurrentValues مقادیر جاری و تغییر یافته موجودیت را باز می‌گرداند. به علاوه این خروجی امکان فراخوانی متد GetDatabaseValues را جهت بدست آوردن مقادیر جدید ذخیره شده در بانک اطلاعاتی نیز ارائه می‌دهد. ممکن است در این بین، خارج از Context جاری، اطلاعات بانک اطلاعاتی توسط کاربر دیگری تغییر کرده باشد. به کمک GetDatabaseValues می‌توان به این نوع اطلاعات نیز دست یافت.
حداقل چهار کاربرد عملی جالب را از اطلاعات موجود در Change Tracker API می‌توان مثال زد که در ادامه به بررسی آن‌ها خواهیم پرداخت.

کلاس‌های مدل مثال جاری

در اینجا یک رابطه many-to-one بین جدول هزینه‌های اقلام خریداری شده یک شخص و جدول فروشندگان تعریف شده است:

```
using System;

namespace EF_Sample09.DomainClasses
{
    public abstract class BaseEntity
    {
        public int Id { get; set; }

        public DateTime CreatedOn { set; get; }
        public string CreatedBy { set; get; }

        public DateTime ModifiedOn { set; get; }
        public string ModifiedBy { set; get; }
    }
}
```

```
using System;

namespace EF_Sample09.DomainClasses
{
    public class Bill : BaseEntity
    {
        public decimal Amount { set; get; }
        public string Description { get; set; }

        public virtual Payee Payee { get; set; }
    }
}
```

```
using System.Collections.Generic;

namespace EF_Sample09.DomainClasses
{
    public class Payee : BaseEntity
    {
        public string Name { get; set; }

        public virtual ICollection<Bill> Bills { set; get; }
    }
}
```

به علاوه همانطور که ملاحظه می‌کنید، این کلاس‌ها از یک abstract class به نام BaseEntity مشتق شده‌اند. هدف از این کلاس پایه تنها تامین یک سری خواص تکراری در کلاس‌های برنامه است و هدف از آن، مباحث ارث بری مانند TPH، TPT و TPC نیست.
به همین جهت برای اینکه این کلاس پایه تبدیل به یک جدول مجزا و یا سبب یکی شدن تمام کلاس‌ها در یک جدول نشود، تنها کافی است آن‌را به عنوان DbSet معرفی نکنیم و یا می‌توان از متد Ignore نیز استفاده کرد:

```
using System.Data.Entity;
using EF_Sample09.DomainClasses;

namespace EF_Sample09.DataLayer.Context
{
    public class Sample09Context : MyDbContextBase
    {
        public DbSet<Bill> Bills { set; get; }
        public DbSet<Payee> Payees { set; get; }

        protected override void OnModelCreating(DbModelBuilder modelBuilder)
        {
            modelBuilder.Ignore<BaseEntity>();

            base.OnModelCreating(modelBuilder);
        }
    }
}
```

الف) به روز رسانی اطلاعات Context در صورتیکه از متد context.Database.ExecuteSqlCommand مستقیما استفاده شود

در قسمت قبل با متد context.Database.ExecuteSqlCommand برای اجرای مستقیم عبارات SQL بر روی بانک اطلاعاتی آشنا شدیم. اگر این متد در نیمه کار یک Context فراخوانی شود، به معنای کنار گذاشتن Change Tracker API می‌باشد؛ زیرا اکنون در سمت بانک اطلاعاتی اتفاقاتی رخ داده‌اند که هنوز در Context جاری کلاینت منعکس نشده‌اند:

```
using System;
using System.Data.Entity;
using EF_Sample09.DataLayer.Context;
using EF_Sample09.DomainClasses;

namespace EF_Sample09
{
    class Program
    {
        static void Main(string[] args)
        {
            Database.SetInitializer(new MigrateDatabaseToLatestVersion<Sample09Context,
Configuration>());

            using (var db = new Sample09Context())
            {
                var payee = new Payee { Name = "فروشگاه سر کوچه" };
```

```
                var bill = new Bill { Amount = 4900, Description = "یک سطل ماست", Payee = payee };
                db.Bills.Add(bill);

                db.SaveChanges();
            }

            using (var db = new Sample09Context())
            {
                var bill1 = db.Bills.Find(1);
                bill1.Description = "ماست";

                db.Database.ExecuteSqlCommand("Update Bills set Description=N'سطل ماست' where
id=1");
                Console.WriteLine(bill1.Description);

                db.Entry(bill1).Reload(); //Refreshing an Entity from the Database
                Console.WriteLine(bill1.Description);

                db.SaveChanges();
            }
        }
    }
}
```

در این مثال ابتدا دو رکورد به بانک اطلاعاتی اضافه می‌شوند. سپس توسط متد db.Bills.Find، اولین رکورد جدول Bills بازگشت داده می‌شود. در ادامه، خاصیت توضیحات آن به روز شده و سپس با استفاده از متد db.Database.ExecuteSqlCommand نیز بار دیگر خاصیت توضیحات اولین رکورد به روز خواهد شد.
اکنون اگر مقدار bill1.Description را بررسی کنیم، هنوز دارای مقدار پیش از فراخوانی db.Database.ExecuteSqlCommand می‌باشد، زیرا تغییرات سمت بانک اطلاعاتی هنوز به Context مورد استفاده منعکس نشده است.
در اینجا برای هماهنگی کلاینت با بانک اطلاعاتی، کافی است متد Reload را بر روی موجودیت مورد نظر فراخوانی کنیم.

ب) یکسان سازی ی و ک اطلاعات رشته‌ای دریافتی پیش از ذخیره سازی در بانک اطلاعاتی

یکی از الزامات برنامه‌های فارسی، یکسان سازی ی و ک دریافتی از کاربر است. برای این منظور باید پیش از فراخوانی متد SaveChanges نهایی، مقادیر رشته‌ای کلیه موجودیت‌ها را یافته و به روز رسانی کرد:

```
using System;
using System.Data;
using System.Data.Entity;
using System.Linq;
using System.Reflection;
using EF_Sample09.DataLayer.Toolkit;
using EF_Sample09.DomainClasses;

namespace EF_Sample09.DataLayer.Context
{
    public class MyDbContextBase : DbContext
    {
        public void RejectChanges()
        {
            foreach (var entry in this.ChangeTracker.Entries())
            {
                switch (entry.State)
                {
                    case EntityState.Modified:
                        entry.State = EntityState.Unchanged;
                        break;

                    case EntityState.Added:
                        entry.State = EntityState.Detached;
                        break;
                }
            }
        }

        public override int SaveChanges()
        {
```

```
            applyCorrectYeKe();
            auditFields();
            return base.SaveChanges();
        }

        private void applyCorrectYeKe()
        {
            //پیدا کردن موجودیت‌های تغییر کرده
            var changedEntities = this.ChangeTracker
                                      .Entries()
                                      .Where(x => x.State == EntityState.Added || x.State ==
EntityState.Modified);

            foreach (var item in changedEntities)
            {
                if (item.Entity == null) continue;

                //یافتن خواص قابل تنظیم و رشته‌ای این موجودیت‌ها
                var propertyInfos = item.Entity.GetType().GetProperties(
                    BindingFlags.Public | BindingFlags.Instance
                    ).Where(p => p.CanRead && p.CanWrite && p.PropertyType == typeof(string));

                var pr = new PropertyReflector();

                //اعمال یکپارچگی نهایی
                foreach (var propertyInfo in propertyInfos)
                {
                    var propName = propertyInfo.Name;
                    var val = pr.GetValue(item.Entity, propName);
                    if (val != null)
                    {
                        var newVal = val.ToString().Replace("ی" ,"ی").Replace("ک" ,"ک");
                        if (newVal == val.ToString()) continue;
                        pr.SetValue(item.Entity, propName, newVal);
                    }
                }
            }
        }

        private void auditFields()
        {
            // var auditUser = User.Identity.Name; // in web apps
            var auditDate = DateTime.Now;
            foreach (var entry in this.ChangeTracker.Entries<BaseEntity>())
            {
                // Note: You must add a reference to assembly : System.Data.Entity
                switch (entry.State)
                {
                    case EntityState.Added:
                        entry.Entity.CreatedOn = auditDate;
                        entry.Entity.ModifiedOn = auditDate;
                        entry.Entity.CreatedBy = "auditUser";
                        entry.Entity.ModifiedBy = "auditUser";
                        break;

                    case EntityState.Modified:
                        entry.Entity.ModifiedOn = auditDate;
                        entry.Entity.ModifiedBy = "auditUser";
                        break;
                }
            }
        }
    }
}
```

اگر به کلاس Context مثال جاری که در ابتدای بحث معرفی شد دقت کرده باشید به این نحو تعریف شده است (بجای DbContext از MyDbContextBase مشتق شده):

```
public class Sample09Context : MyDbContextBase
```

علت هم این است که یک سری کد تکراری را که می‌توان در تمام Contextها قرار داد، بهتر است در یک کلاس پایه تعریف کرده و سپس از آن ارث بری کرد.

تعاریف کامل کلاس MyDbContextBase را در کدهای فوق ملاحظه می‌کنید.

در اینجا کار با تحریف متد SaveChanges شروع می‌شود. سپس در متد applyCorrectYeKe کلیه موجودیت‌های تحت نظر ChangeTracker که تغییر کرده باشند یا به آن اضافه شده باشند، یافت شده و سپس خواص رشته‌ای آن‌ها جهت یکسانی سازی ی و ک، بررسی می‌شوند.

ج) ساده‌تر سازی به روز رسانی فیلدهای بازبینی یک رکورد مانند DateCreated، DateLastUpdated و امثال آن بر اساس وضعیت جاری یک موجودیت

در کلاس MyDbContextBase فوق، کار متد auditFields، مقدار دهی خودکار خواص تکراری تاریخ ایجاد، تاریخ به روز رسانی، شخص ایجاد کننده و شخص تغییر دهنده یک رکورد است. به کمک ChangeTracker می‌توان به موجودیت‌هایی از نوع کلاس پایه BaseEntity دست یافت. در اینجا اگر entry.State آن‌ها مساوی EntityState.Added بود، هر چهار خاصیت یاد شده به روز می‌شوند. اگر حالت موجودیت جاری، EntityState.Modified بود، تنها خواص مرتبط با تغییرات رکورد به روز خواهند شد.
به این ترتیب دیگر نیازی نیست تا در حین ثبت یا ویرایش اطلاعات برنامه نگران این چهار خاصیت باشیم؛ زیرا به صورت خودکار مقدار دهی خواهند شد.

د) پیاده سازی قابلیت لغو تغییرات در برنامه

علاوه بر این‌ها در کلاس MyDbContextBase، متد RejectChanges نیز تعریف شده است تا بتوان در صورت نیاز، حالت موجودیت‌های تغییر کرده یا اضافه شده را به حالت پیش از عملیات، بازگرداند.

عنوان: EF Code First #15
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۲/۳۰ ۱۲:۲۰:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Entity framework

EF Code first و بانک‌های اطلاعاتی متفاوت

در آخرین قسمت از سری EF Code first بد نیست نحوه استفاده از بانک‌های اطلاعاتی دیگری را بجز SQL Server نیز بررسی کنیم. در اینجا کلاس‌های مدل و کدهای مورد استفاده نیز همانند قسمت 14 است و تنها به ذکر تفاوت‌ها و نکات مرتبط اکتفاء خواهد شد.

حالت کلی پشتیبانی از بانک‌های اطلاعاتی مختلف توسط EF Code first

EF Code first با کلیه پروایدرهای تهیه شده برای ADO.NET 3.5 که پشتیبانی از EF را لحاظ کرده باشند، به خوبی کار می‌کند. پروایدرهای مخصوص ADO.NET 4.0، تنها سه گزینه DeleteDatabase/CreateDatabase/DatabaseExists را نسبت به نگارش قبلی بیشتر دارند و EF Code first ویژگی‌های بیشتری را طلب نمی‌کند.
بنابراین اگر حین استفاده از پروایدر ADO.NET مخصوص بانک اطلاعاتی خاصی با پیغام «CreateDatabase is not supported by the provider» مواجه شدید، به این معنا است که این پروایدر برای دات نت 4 به روز نشده است. اما به این معنا نیست که با EF Code first کار نمی‌کند. فقط باید یک دیتابیس خالی از پیش تهیه شده را به برنامه معرفی کنید تا مباحث Database Migrations به خوبی کار کنند؛ یا اینکه کلا می‌توانید Database Migrations را خاموش کرده (متد Database.SetInitializer را با پارامتر نال فراخوانی کنید) و فیلدها و جداول را دستی ایجاد کنید.

استفاده از EF Code first با SQLite

برای استفاده از SQLite در دات نت ابتدا نیاز به پروایدر ADO.NET آن است: « مکان دریافت درایورهای جدید SQLite مخصوص دات نت »
ضمن اینکه به نکته « استفاده از اسمبلی‌های دات نت 2 در یک پروژه دات نت 4 » نیز باید دقت داشت.
و یکی از بهترین management studio هایی که برای آن تهیه شده: « SQLite Manager »
پس از دریافت پروایدر آن، ارجاعی را به اسمبلی System.Data.SQLite.dll به برنامه اضافه کنید.
سپس فایل کانفیگ برنامه را به نحو زیر تغییر دهید:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <section name="entityFramework"
type="System.Data.Entity.Internal.ConfigFile.EntityFrameworkSection, EntityFramework, Version=4.3.1.0,
Culture=neutral, PublicKeyToken=b77a5c561934e089" />
  </configSections>
  <startup useLegacyV2RuntimeActivationPolicy="true">
    <supportedRuntime version="v4.0"/>
  </startup>

  <connectionStrings>
    <clear/>
    <add name="Sample09Context"
         connectionString="Data Source=CodeFirst.db"
         providerName="System.Data.SQLite"/>
  </connectionStrings>
</configuration>
```

همانطور که ملاحظه می‌کنید، تفاوت آن با قبل، تغییر connectionString و providerName است.

اکنون اگر همان برنامه قسمت قبل را اجرا کنیم به خطای زیر برخواهیم خورد:
«The given key was not present in the dictionary»
در این مورد هم توضیح داده شد. سه گزینه DeleteDatabase/CreateDatabase/DatabaseExists در پروایدر جاری SQLite برای دات نت وجود ندارد. به همین جهت نیاز است فایل «CodeFirst.db» ذکر شده در کانکشن استرینگ را ابتدا دستی درست کرد.
برای مثال از افزونه SQLite Manager استفاده کنید. ابتدا یک بانک اطلاعاتی خالی را درست کرده و سپس دستورات زیر را بر روی بانک اطلاعاتی اجرا کنید تا دو جدول خالی را ایجاد کند (در برگه Execute sql افزونه SQLite Manager):

```
CREATE TABLE [Payees](
 [Id] [integer] PRIMARY KEY AUTOINCREMENT NOT NULL,
 [Name] [text] NULL,
 [CreatedOn] [datetime] NOT NULL,
 [CreatedBy] [text] NULL,
 [ModifiedOn] [datetime] NOT NULL,
 [ModifiedBy] [text] NULL
);

CREATE TABLE [Bills](
 [Id] [integer] PRIMARY KEY AUTOINCREMENT NOT NULL,
 [Amount] [float](18, 2) NOT NULL,
 [Description] [text] NULL,
 [CreatedOn] [datetime] NOT NULL,
 [CreatedBy] [text] NULL,
 [ModifiedOn] [datetime] NOT NULL,
 [ModifiedBy] [text] NULL,
 [Payee_Id] [integer] NULL
);
```

سپس سطر زیر را نیز به ابتدای برنامه اضافه کنید:

```
Database.SetInitializer<Sample09Context>(null);
```

به این ترتیب database migrations خاموش می‌شود و اکنون برنامه بدون مشکل کار خواهد کرد.
فقط باید به یک سری نکات مانند نوع داده‌ها در بانک‌های اطلاعاتی مختلف دقت داشت. برای مثال integer در اینجا از نوع Int64 است؛ بنابراین در برنامه نیز باید به همین ترتیب تعریف شود تا نگاشت‌ها به درستی انجام شوند.

در کل تنها مشکل پروایدر فعلی SQLite عدم پشتیبانی از مباحث database migrations است. این مورد را خاموش کرده و تغییرات ساختار بانک اطلاعاتی را به صورت دستی به بانک اطلاعاتی اعمال کنید. بدون مشکل کار خواهد کرد.

البته اگر به دنبال پروایدری تجاری با پشتیبانی از آخرین نگارش EF Code first هستید، گزینه زیر نیز مهیا است:
/http://devart.com/dotconnect/sqlite
برای مثال اگر علاقمند به استفاده از حالت تشکیل بانک اطلاعاتی SQLite در حافظه هستید (با رشته اتصالی ویژه Data Source=:memory:;Version=3;New=True;)، فعلا تنها گزینه مهیا استفاده از پروایدر تجاری فوق است؛ زیرا مبحث Database Migrations را به خوبی پشتیبانی می‌کند.

استفاده از EF Code first با SQL Server CE

قبلا در مورد « استفاده از SQL-CE به کمک NHibernate » مطلبی را در این سایت مطالعه کرده‌اید. سه مورد اول آن با EF Code first یکی است و تفاوتی نمی‌کند (یک سری بحث عمومی مشترک است). البته با یک تفاوت؛ در اینجا EF Code first قادر است یک بانک اطلاعاتی خالی SQL Server CE را به صورت خودکار ایجاد کند و نیازی نیست تا آن‌را دستی ایجاد کرد. مباحث database migrations و به روز رسانی خودکار ساختار بانک اطلاعاتی نیز در اینجا پشتیبانی می‌شود.

برای استفاده از آن ابتدا ارجاعی را به اسمبلی System.Data.SqlServerCe.dll قرار گرفته در مسیر Program Files\Microsoft SQL Server Compact Edition\v4.0\Desktop اضافه کنید.
سپس رشته اتصالی به بانک اطلاعاتی و providerName را به نحو زیر تغییر دهید:

```
<connectionStrings>
    <clear/>
    <add name="Sample09Context"
         connectionString="Data Source=mydb.sdf;Password=1234;Encrypt Database=True"
         providerName="System.Data.SqlServerCE.4.0"/>
</connectionStrings>
```

بدون نیاز به هیچگونه تغییری در کدهای برنامه، همین مقدار تغییر در تنظیمات ابتدایی برنامه برای کار با SQL Server CE کافی است.
ضمنا مشکلی هم با فیلد Identity در آخرین نگارش EF Code first وجود ندارد؛ برخلاف حالت database first آن که پیشتر این اجازه را نمی‌داد و خطای «Server-generated keys and server-generated values are not supported by SQL Server Compact» را ظاهر می‌کرد.

استفاده از EF Code first با MySQL

برای استفاده از EF Code first با MySQL (نگارش 5 به بعد البته) ابتدا نیاز است پروایدر مخصوص ADO.NET آن‌را دریافت کرد: (^)
که از EF نیز پشتیبانی می‌کند. پس از نصب آن، ارجاعی را به اسمبلی MySql.Data.dll قرار گرفته در مسیر Program Files\MySQL\MySQL Connector Net 6.5.4\Assemblies\v4.0 به پروژه اضافه نمائید.
سپس رشته اتصالی و providerName را به نحو زیر تغییر دهید:

```
<connectionStrings>
    <clear/>
    <add name="Sample09Context"
         connectionString="Datasource=localhost; Database=testdb2; Uid=root; Pwd=123;"
         providerName="MySql.Data.MySqlClient"/>
</connectionStrings>

<system.data>
    <DbProviderFactories>
      <remove invariant="MySql.Data.MySqlClient"/>
      <add name="MySQL Data Provider"
           invariant="MySql.Data.MySqlClient"
           description=".Net Framework Data Provider for MySQL"
           type="MySql.Data.MySqlClient.MySqlClientFactory, MySql.Data, Version=6.5.4.0,
Culture=neutral, PublicKeyToken=c5687fc88969c44d" />
    </DbProviderFactories>
</system.data>
```

همانطور که مشاهده می‌کنید در اینجا شماره نگارش دقیق پروایدر مورد استفاده نیز ذکر شده است. برای مثال اگر چندین پروایدر روی سیستم نصب است، با مقدار دهی DbProviderFactories می‌توان از نگارش مخصوصی استفاده کرد.

با این تغییرات پس از اجرای برنامه قسمت قبل، به خطای زیر برخواهیم خورد:
The given key was not present in the dictionary

توضیحات این مورد با قسمت SQLite یکی است؛ به عبارتی نیاز است بانک اطلاعاتی testdb را دستی درست کرد. همچنین

جداول و فیلدها را نیز باید دستی ایجاد کرد و database migrations را نیز باید خاموش کرد (پارامتر Database.SetInitializer را به نال مقدار دهی کنید).
برای این منظور یک دیتابیس خالی را ایجاد کرده و سپس دو جدول زیر را به آن اضافه کنید:

```
CREATE TABLE IF NOT EXISTS `bills` (
  `Id` int(11) NOT NULL AUTO_INCREMENT,
  `Amount` float DEFAULT NULL,
  `Description` varchar(400) CHARACTER SET utf8 COLLATE utf8_persian_ci NOT NULL,
  `CreatedOn` datetime NOT NULL,
  `CreatedBy` varchar(400) CHARACTER SET utf8 COLLATE utf8_persian_ci NOT NULL,
  `ModifiedOn` datetime NOT NULL,
  `ModifiedBy` varchar(400) CHARACTER SET utf8 COLLATE utf8_persian_ci NOT NULL,
  `Payee_Id` int(11) NOT NULL,
  PRIMARY KEY (`Id`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 COLLATE=utf8_persian_ci AUTO_INCREMENT=1 ;

CREATE TABLE IF NOT EXISTS `payees` (
  `Id` int(11) NOT NULL AUTO_INCREMENT,
  `Name` varchar(400) CHARACTER SET utf8 COLLATE utf8_persian_ci NOT NULL,
  `CreatedOn` datetime NOT NULL,
  `CreatedBy` varchar(400) CHARACTER SET utf8 COLLATE utf8_persian_ci NOT NULL,
  `ModifiedOn` datetime NOT NULL,
  `ModifiedBy` varchar(400) CHARACTER SET utf8 COLLATE utf8_persian_ci NOT NULL,
  PRIMARY KEY (`Id`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8 COLLATE=utf8_persian_ci AUTO_INCREMENT=1 ;
```

پس از این تغییرات، برنامه بدون مشکل اجرا خواهد شد (ایجاد بانک اطلاعاتی خالی به همراه ایجاد ساختار جداول و خاموش کردن database migrations که توسط این پروایدر پشتیبانی نمی‌شود).

به علاوه پروایدر تجاری دیگری هم در سایت devart.com برای MySQL و EF Code first مهیا است که مباحث database migrations را به خوبی مدیریت می‌کند.

مشکل!

اگر به همین نحو برنامه را اجرا کنیم، فیلدهای یونیکد فارسی ثبت شده در MySQL با «??????? ?? ????» مقدار دهی خواهند شد و تنظیم CHARACTER SET utf8 COLLATE utf8_persian_ci نیز کافی نبوده است (این مورد با SQLite یا نگارش‌های مختلف SQL Server بدون مشکل کار می‌کند و نیاز به تنظیم اضافه‌تری ندارد):

```
ALTER TABLE `bills` DEFAULT CHARACTER SET utf8 COLLATE utf8_persian_ci
```

برای رفع این مشکل توصیه شده است که CharSet=UTF8 را به رشته اتصالی به بانک اطلاعاتی اضافه کنیم. اما در این حالت خطای زیر ظاهر می‌شود:

The provider did not return a ProviderManifestToken string

این مورد فقط به اشتباه بودن تعاریف رشته اتصالی بر می‌گردد؛ یا عدم پشتیبانی از تنظیم اضافه‌ای که در رشته اتصالی ذکر شده است.

مقدار صحیح آن دقیقا مساوی CHARSET=utf8 است (با همین نگارش و رعایت کوچکی و بزرگی حروف؛ مهم!):

```
<connectionStrings>
    <clear/>
    <add name="Sample09Context"
         connectionString="Datasource=localhost; Database=testdb; Uid=root; Pwd=123;CHARSET=utf8"
         providerName="MySql.Data.MySqlClient"/>
</connectionStrings>
```

به این ترتیب، مشکل ثبت عبارات یونیکد فارسی برطرف می‌شود (البته جدول هم بهتر است به DEFAULT CHARACTER SET utf8 COLLATE utf8_persian_ci تغییر پیدا کند؛ مطابق دستور Alter ایی که در بالا ذکر شد).